Enhancing Trust –A Unified Meta-Model for Software Security Vulnerability Analysis

Sultan Saud Alqahtani

A Thesis
In the Department
of
Computer Science and Software Engineering

Presented in Partial Fulfillment of the Requirements
For the Degree of
Doctor of Philosophy (Software Engineering) at
Concordia University
Montreal, Quebec, Canada

July 2018

CONCORDIA UNIVERSITY

School of Graduate Studies

This is to certify that the thesis prepared

By: **Sultan Saud Alqahtani**

Entitled: **Enhancing Trust – A Unified Meta-Model for Software Security Vulnerability Analysis**

and submitted in partial fulfillment of the requirements for the degree of

Doctor of Philosophy (Computer Science)

complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the final examining committee:

______________________________ Chair
Dr. Rolf Wüthrich

______________________________ External Examiner
Dr. Ettore Merlo

______________________________ Examiner
Dr. Wahab Hamou-Lhadj

______________________________ Examiner
Dr. Todd Eavis

______________________________ Examiner
Dr. Nikolaos Tsantalis

______________________________ Supervisor
Dr. Juergen Rilling

Approved by ______________________________
Dr. Volker Haarslev, Graduate Program Director

30 August 2018 ______________________________
Dr. Amir Asif, Dean

Faculty of Engineering and Computer Science

Abstract

Enhancing Trust – A Unified Meta-Model for Software Security Vulnerability Analysis

Sultan Saud Alqahtani, Ph.D.

Concordia University, 2018

Over the last decade, a globalization of the software industry has taken place which has 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 code implementation being shared across systems but also any vulnerabilities it is exposed to as well. Hence, vulnerabilities found in APIs no longer affect only individual projects but instead might spread across projects and even global software ecosystem borders. Tracing such vulnerabilities on a global scale becomes an inherently difficult task, with many of the resources required for the analysis not only growing at unprecedented rates but also being spread across heterogeneous resources. Software developers are struggling to identify and locate the required data to take full advantage of these resources. The Semantic Web and its supporting technology stack have been widely promoted to model, integrate, and support interoperability among heterogeneous data sources.

This dissertation introduces four major contributions to address these challenges: (1) It provides a literature review of the use of software vulnerabilities databases (SVDBs) in the Software Engineering community. (2) Based on findings from this literature review, we present SEVONT, a Semantic Web based modeling approach to support a formal and semi-automated approach for unifying vulnerability information resources. SEVONT introduces a multi-layer knowledge model which not only provides a unified knowledge representation, but also captures software vulnerability information at different abstract levels to allow for seamless integration, analysis, and reuse of the modeled knowledge. The modeling approach takes advantage of Formal Concept Analysis (FCA) to guide knowledge engineers in identifying reusable knowledge concepts and modeling them. (3) A Security Vulnerability Analysis Framework (SV-AF) is introduced, which is an instantiation of the SEVONT knowledge model to support evidence-based vulnerability detection. The framework integrates vulnerability ontologies (and data) with existing Software Engineering ontologies allowing for the use of Semantic Web reasoning services to trace and assess the impact of security vulnerabilities across project boundaries.

Several case studies are presented to illustrate the applicability and flexibility of our modelling approach, demonstrating that the presented knowledge modeling approach cannot only unify heterogeneous vulnerability data sources but also enables new types of vulnerability analysis.

DEDICATION

To my father’s mother (my grandmother Naffla bint Saad) of blessed memory, who passed away 40 days before the oral defense of this thesis.

Without her sacrifices, my life as a PhD student would not have been a reality. Unfortunately, she never got to see me become the first “Dr.” in my family. I hope that she can see me now and that I have her admiration as she has mine. May her memories be only for a blessing.

To my parents for their endless love, support, and encouragement.

ACKNOWLEDGEMENTS

First and foremost, all praises to Allah for blessing, protecting, and guiding me throughout my studies. I could never have accomplished this without my faith.

This thesis is dedicated to my grandmother of blessed memory, whom I wish to formally acknowledge here. My grandmother Nafla bint Saad, the matriarch of our family, passed away 40 days before I completed the oral examination for this thesis. For many months, my grandma put up with my exceptional stress and busyness as this thesis winded down and she saw much less of me than usual. Although she was not always skilled at hiding her disappointment when I was too busy to visit, she took great pride in knowing that soon I would be done with my doctorate. Grandma, I'm done.

I would like to express my sincere gratitude to my supervisor Prof. Juergen Rilling for his continuous support, patience, motivation, and immense knowledge. His unique personality as a supervisor and friend is the main reason behind the success of this research. The research objectives would not have been achieved without the professional and experienced guidance and support of my supervisor. I also extend my thanks to members of the examining committee including Dr. Wahab Hamou-Lhadj, Dr. Todd Eavis, Dr. Nikolaos Tsantalis, and Dr. Ettore Merlo (External) for critically evaluating my thesis and providing me with invaluable comments at different stages of my research.

My gratitude also goes to Ambient Software Evolution Group members at Concordia University: Rabe Abdalkareem, Mostafa Erfani, Ellis Emmanuel Eghan, SayedHassan Khatoonabadi, Chris Forbes, Parisa Moslehi, and Yasaman Sarlati. Special thanks go to the incredible Ellis Emmanuel Eghan for his help as well as his time spent working on our first project (Semantic Global Problem Scanner – SEGPS). Through numerous emails and Skype conferences, I have been lucky to have a colleague like Ellis who has the ability to tackle big problems and make them easy to solve. I am forever thankful to my brothers and colleagues Hamad BinSalleeh and Feras Aljumah, whose friendship made this journey and my stay in Canada truly memorable. I would also like to acknowledge the financial support from the Government of Saudi Arabia under the scholarship of Imam Mohammed Bin Saud University, which enabled me to undertake my PhD studies.

Last but certainly never least, I take this opportunity to express my deepest gratitude to my first teachers (my dear mother and my father), my brothers, and my sisters for their unconditional love, prayers, and support. Without them, this journey would not have been possible, and to them I dedicate this milestone.

Table of Contents

List of Figures

List of Tables

List of Acronyms

CAPEC	Common Attack Pattern Enumeration and Classification
CPE	Common Platform Enumeration
CVE	Common Vulnerabilities Exposure
CVSS	Common Vulnerabilities Scoring System
CWE	Common Weakness Enumeration
DL	Description Logic
FCA	Formal Concept Analysis
NVD	National Vulnerability Database
OSS	Open Source Software
OWL	The Web Ontology Language
PSL	Probabilistic Soft Logic
RDF	Resource Description Format
SBSON	Software Build Systems Ontologies
SE	Software Engineering
SEON	Software Evolution ONtologies
SEVONT	Software sEcurity Vulnerability ONTology
SVDB	Security Vulnerability Database
SW	The Semantic Web
WVD	Weighted Vulnerability Density

Chapter 1

1 Introduction

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

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

Shared knowledge resources not only facilitate reuse and collaboration; they also introduce new challenges to the Software Engineering (SE) community. Knowledge and resources are no longer controlled by a single project or organization, but instead are now distributed across multiple projects, organizations, or even global software ecosystems. Among the challenges arising from this knowledge sharing is Information Security (IS), which has emerged as a major threat to the software development community. At its core, IS promotes the notion that one should consider different security concepts (e.g., secure coding practices, knowledge about

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

software security vulnerabilities and their analysis) in the development process. The importance of IS for the software community is reflected by the fact that it has become an integrated part of current SE best practices [4].

As part of this IS integration process, developers and other stakeholders require access to vulnerability related knowledge resources. Security vulnerabilities databases (SVDBs) are an example of such knowledge resources which can provide software professionals with access to existing security issues affecting different software products. Access to this information can help prevent inadvertent (security) mistakes that might damage their systems or reduce the potential risk their systems might be exposed to. Different public SVDBs (e.g., National Vulnerability Database - NVD[4]) have been introduced to track known software vulnerabilities and potential solutions to resolve them. These SVDBs can be seen as a direct response by the software industry to the ever-increasing number of software attacks, which are no longer limited to a particular project or computer but now often affect hundreds of different systems and millions of computers.

1.1 Problem Statement

In response to the increasing number of software vulnerabilities and attacks, various private and public organizations have introduced SVDBs. Each of these databases captures not only different types of vulnerability information but are also of interest to developers and other stakeholders since they contain valuable information about software flaws, causes of defects, and details on how vulnerabilities arise, etc. [4].

While the SE research community is becoming increasingly aware of these SVDBs, no comprehensive literature survey exists that studies how they (i.e., SVDBs) are used in software development.

Problem Statement 1: Investigating the SE literature will provide insights on how usage of SVDBs in the SE domain has evolved over the past decade and outline some open challenges associated with their use.

[4] https://nvd.nist.gov/

Effective use of security vulnerability information can enhance software productivity, create economic benefit to software stakeholders (reduce cost and consuming time), and increase security of the software systems. However, with security vulnerability data growing at unprecedented rates, SE stakeholders (e.g., developers) are struggling to take full advantage of available vulnerabilities information. One main reason for this is that vulnerabilities data usually originates from disparate sources. This can result in vulnerabilities data heterogeneity and prevent data from being digested easily. Among other techniques developed, Semantic Web technologies and its supporting ontology based approaches are a promising method for overcoming heterogeneity and improving vulnerability data interoperability.

Problem Statement 2: Providing a unified knowledge-model for different vulnerabilities sources (e.g., SVDBs) can be a viable solution to integrate existing vulnerability data sources.

SE developers are often unaware or unfamiliar with SVDBs and the fact that known vulnerabilities might already be published in these SVDBs. As a result, vulnerabilities affecting their systems (either directly or indirectly through external vulnerable components used by their systems) often remain uncovered. For example, in September 2017, the Equifax breach [5] was caused by the Apache Struts[5] vulnerability [6]. The Apache Struts vulnerability CVE-2017-5638[6] was disclosed on March 7 and patched on the very same day, meaning a secured version of Apache Struts was available in SVDBs since March 7 for developers to update any vulnerable version they might have.

However, increasing both awareness and automate accessibility to SVDBs during software development can improve software reliability and quality. Establishing such required traceability among vulnerabilities across software artifacts is an essential aspect in identifying and locating vulnerable code, applying existing fixes, and improving the analysis of potential impacts of vulnerabilities.

In addition, new vulnerabilities are constantly being found in OSS code and many projects have no mechanisms in place for locating and fixing these problems. According to a recent Snyk survey [7] of open source maintainers, 44% have never had a security audit and only 17% said that they had a high level of security know-how. The survey also showed that 34% of the

[5] Apache Struts is a popular open-source framework for developing web applications in the Java programming language.

[6] https://nvd.nist.gov/vuln/detail/CVE-2017-5638

developers surveyed indicated that they use deprecate, the older, insecure version. Twenty five percent reported that they make no effort at all to notify users of vulnerabilities, and only 10% file a CVE[7].

A survey conducted by Sonatype [8] further suggests that software reuse through OSS components has become a de-facto industry norm, with 90% of survey participants relying on pre-existing open source code in their own implementation. Moreover, deployed OSS polices are reported to have major shortcomings with only 21% of these policies enforcing the use of secure OSS code, 63% having no active monitoring of vulnerabilities over time, and lastly, 78% of the surveyed policies have never banned the usage of certain OSS components. This poor management of third party artifacts in software systems could be compromised by intruders.

As the use of OSS grows, this risk surface expands. A recent report [9] reveals that there is no standard way of documenting security on OSS projects. In the top 400,000 public repositories on GitHub, only 2.4% had security documentation in place.

Problem Statement 3: With known vulnerabilities published in public SVDBs and using these SVDBs knowledge model in software development, managing the security of OSS components, tracing the vulnerabilities impacts, and calculating the OSS components trustworthiness can be automated.

In the next section, we summarize the specific thesis contributions and how these contributions address the problem statements.

1.2 Thesis Contributions

In this thesis, we make the following contributions:

- We conduct a literature survey; we reviewed 94 articles discussing the use of SVDBs in the SE domain (Chapter 3). Among the key findings of this survey are:
    - An increasing awareness in the research community that describes the use and application of SVDBs in the SE domain.

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

    - The majority of the surveyed papers apply SVDBs only to a limited number of SE activities.
    - Most contributions only rely on a single SVDB in their approach.
- We introduce a novel knowledge engineering methodology that applies Formal Concept Analysis (FCA) to semi-automate the software vulnerabilities knowledge acquisition and extraction from SVDBs (Chapter 4).
    - We conduct a literature review of existing software security vulnerability ontologies.
    - We propose a semi-automated methodology using FCA to create a unified ontological knowledge model (SEVONT, **S**oftware s**E**curity **V**ulnerability **ONT**ology) that supports knowledge sharing, linking, and inference across SVDBs boundaries.
    - We present alignment rules to facilitate knowledge integration and improve our overall knowledge design.
    - We illustrate the applicability of our modeling approach by providing examples of how our modeling approach supports vulnerability analysis across individual SVDBs.
- We developed a Security Vulnerability Analysis Framework (SV-AF) to support evidence-based vulnerability detection (Chapter 5). The main contributions of this framework are:
    - Integration of different ontologies such as builds systems ontologies, source code ontologies, version systems ontologies, vulnerabilities ontologies, etc.
    - Applying ontologies alignment using Probabilistic Soft Logic (PSL) to establish weighted links between ontologies.
    - Performed two case studies to illustrate the applicability of the presented approach. We identify that 750 Maven project releases are directly affected by known security vulnerabilities, and by considering transitive dependencies an additional 415,604 Maven projects can be identified as potentially affected by these vulnerabilities.
- Our approach takes advantage of the Semantic Web and its reasoning services, to trace and assess the impact of security vulnerabilities across project boundaries (Chapter 6).

- We introduce a novel Ontological Trustworthiness Assessment Model (OntTAM) which (1) supports the automated analysis and assessment of quality attributes related to the trustworthiness of libraries and APIs in open-source systems, and (2) provides developers with additional insights on the potential impact of reused libraries and APIs on the quality and trustworthiness of their project. We illustrate the applicability of our approach by assessing the trustworthiness of libraries in terms of their API breaking changes, security vulnerabilities, license violations, and their potential impact on client projects (Chapter 7).

1.3 Related Publications

Earlier versions of the work completed in this thesis have been published in the following papers:

1- **S. S. Alqahtani** and J. Rilling, "An Ontology-Based Approach to Automate Tagging of Software Artifacts," 2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM), Toronto, ON, 2017, pp. 169-174.

2- **S. S. Alqahtani**, "Enhancing Trust – Software Vulnerability Analysis Framework," 2017 IEEE International Conference on Software Testing, Verification and Validation (ICST), Tokyo, 2017, pp. 563-564.

3- **S. S. Alqahtani**, E. E. Eghan and J. Rilling, "Recovering Semantic Traceability Links between APIs and Security Vulnerabilities: An Ontological Modeling Approach," 2017 IEEE International Conference on Software Testing, Verification and Validation (ICST), Tokyo, 2017, pp. 80-91.

4- **S. S. Alqahtani**, E. E. Eghan and J. Rilling, "SV-AF — A Security Vulnerability Analysis Framework," 2016 IEEE 27th International Symposium on Software Reliability Engineering (ISSRE), Ottawa, ON, 2016, pp. 219-229.

5- **Sultan S. Alqahtani**, Ellis E. Eghan, Juergen Rilling, “Tracing known security vulnerabilities in software repositories – A Semantic Web enabled modeling approach”, *Science of Computer Programming*, Volume 121, 2016, pp. 153-175.

1.4 Thesis Organization

The thesis is organized as follows. Chapter 2 introduces core concepts and terminologies used in this research. Chapters 3, 4, 5, 6, and 7 are dedicated to the main contributions of this thesis, which were mentioned earlier. The conclusions and some promising avenues for future work are discussed in Chapter 8.

Chapter 2

2 Background

The work presented in this research combines different aspects of multiple fields of study in SE and software security. In this chapter, we provide a brief overview of the core techniques and terminologies used in our research. If you are already familiar with these concepts, you can safely move on to the next chapter as cross-references are provided wherever specific background information is required.

2.1 The Semantic Web and Ontologies in a Nutshell

The term "ontology" originates from philosophy, where it denotes the study of existence. In computer science, a widely accepted definition has been introduced by Studer [10]: "an ontology is a formal, explicit specification of a shared conceptualization." Ontologies are typically used as a formal and explicit way to specify concepts and relationships in a domain of discourse. They can overcome portability, flexibility, and information sharing problems associated with databases [11]. Compared to relational approaches which assume complete knowledge (closed world assumption), ontologies support the modeling of incomplete knowledge (open world assumption) and the extendibility of the ontological model.

The Semantic Web (SW) allows for machine understandable Web resources that can be shared and processed by both software tools (e.g., search engines) and humans [12]. Ontologies are an important foundation of the SW as they allow knowledge to be shared between different agents and the creation of common terminologies for understanding [12]. Moreover, the resulting data representation format becomes reusable rather than being proprietary to specific tasks. The current data-model used to represent meta-data in SW is the Resource Description Format (RDF)

[13]. RDF is used to formalize meta-models in the form of <subject, predicate, object>, which are called triples. RDF triples make statements about resources, with a resource in the SW being anything—a person, project, software, a security bug, etc. In order to make triples persistent, RDF triples stores are used, with each triple being identified by a Uniform Resource Identifier (URI) [13].

The Web Ontology Language (OWL) [14] is used on top of the RDF layer (see Figure 1). It is a standard modeling language put forward by the W3C[8] to pursue the vision of the SW. OWL provides for machine understandable (i.e., capturing semantics) information, allowing Web resources to be automatically processed and integrated. The widely used OWL sub-language OWL-DL is based on Description Logics (DLs) [15].

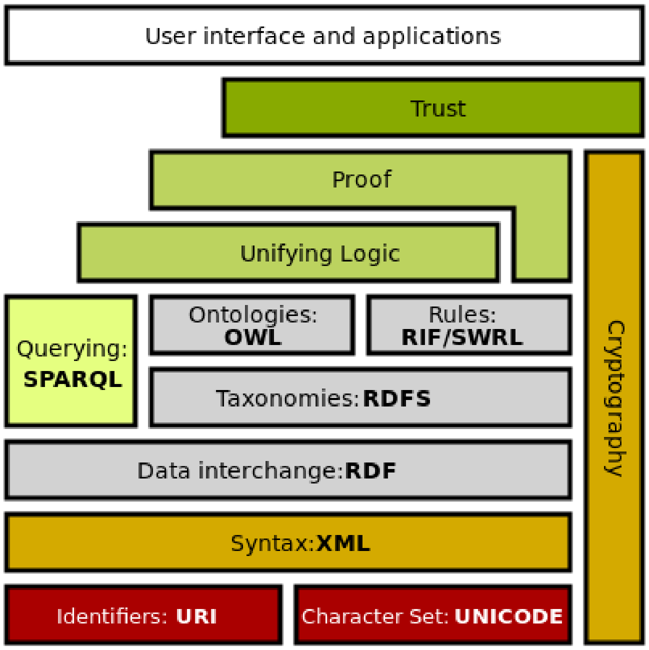


Figure 1: Semantic Web architecture in layers [16].

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

[8] W3C semantic web activity: https://www.w3.org/2001/sw/

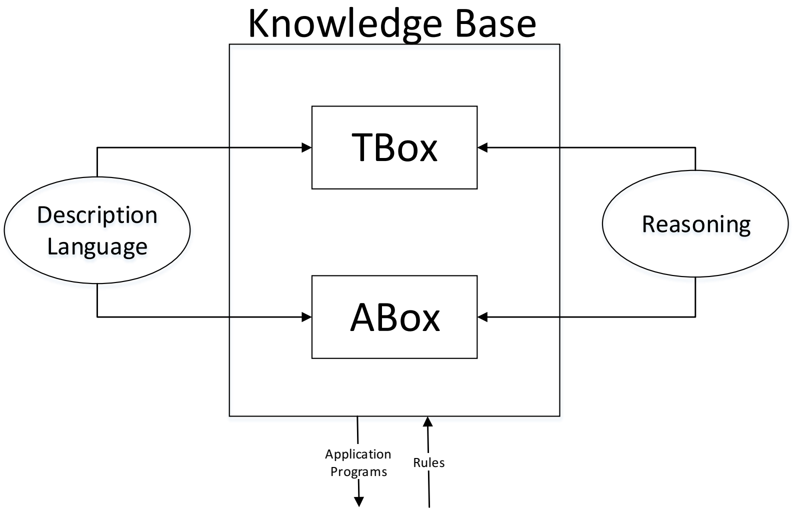


Figure 2: Description Logics System.

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

$$C \sqsubseteq D \ (R \sqsubseteq S) \ or \ C \equiv D \ (R \equiv S)$$

Where C and D are concepts (R and S are roles). The semantics of axioms are defined as: an interpretation $\boldsymbol{I}$ satisfies $C \sqsubseteq D$ $(R \sqsubseteq S)$ if $\boldsymbol{CI} \sqsubseteq \boldsymbol{DI}$ $(\boldsymbol{RI} \sqsubseteq \boldsymbol{SI})$. A Tbox, denoted as $\boldsymbol{T}$, is a finite set of such axioms. The assertions in an ABox are specified using concept assertions and role assertions, which have the form $C(a), R(a, b)$, where C is a concept, R is a role, and a, b are names of individuals. The semantics of assertions can be given as: the interpretation $\boldsymbol{I}$ satisfies the concept assertion $C(a)$ if $\boldsymbol{aI} \in \boldsymbol{CI}$, and it satisfies the role assertion $R(a, b)$ if $(\boldsymbol{aI}, \boldsymbol{bI}) \in \boldsymbol{RI}$. An Abox, denoted as $\boldsymbol{A}$, is a finite set of such assertions.

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

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

In addition to RDF, OWL, and OWL-DL, the SW community provides tools to process OWL semantics and RDF data. Jena [17] emerged as a Java framework for building applications and providing a programmatic environment for RDF and OWL. Reasoners (e.g., RDFS++[9], Pellet[10]) can infer new facts about the designed ontology and form a set of asserted axioms. RDF databases, such as Virtuoso [18] and Allegrograph [19], are used to materialize and store RDF triples. SPARQL is a RDF query language, that is, a semantic query language for databases able to retrieve and manipulate data stored in RDF format.

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

[9] http://franz.com/agraph/support/learning/Overview-of-RDFS++.lhtml
[10] https://www.w3.org/2001/sw/wiki/Pellet

contrasts with relational databases where resources are local and not universal, restricting the ability of relational databases to establish resource links outside their own local schema.

2.2 Ontologies in Software Engineering

Despite ontologies and Knowledge Engineering sharing the same roots, ontologies emphasize aspects such as inter-agent communication and interoperability [21]. 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) [22]. An important aspect of ontologies is that they must be formal and, more precisely, understandable by a computer or "codified in a machine interpretable language" [23].

Ontologies in SE. Representing software in terms of knowledge rather than data, ontologies can be more abstract than, say, database schemata, and provide better support for semantics [13]. With the adoption of Description Logic (DL) as a major foundation of the recently introduced SW and OWL [12], there is a trend to utilize ontologies or introduce taxonomies as conceptual modeling techniques into the SE 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 [24], traceability [25], and use case management [26]. In the software testing domain, KITSS [27] is a knowledge-based system that can provide assistance in converting a semi-formal test case specification into an executable test script. In the software maintenance domain, Ankolekar et al. [28] provide an ontology to model software, developers, and bugs. The authors developed a prototype Semantic Web system, Dhruv, for OSS communities. Dhruv provides an enhanced semantic interface to bug resolution messages and recommends related software objects and artifacts. 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 [29] system allows storing semantic descriptions of components in a knowledge base and performing semantic queries on it. In [30], 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” [31]. However, in SE, ontologies and models try to address the same problems (representing 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 advise developers to look for components that already exist when implementing functionality, since reuse can avoid rework, save money, and improve overall system quality [32]. 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 [33].

2.3 Formal Concept Analysis (FCA)

Formal Concept Analysis (FCA) [34] is a data analysis method that uses mathematical theory to perform data grouping. It provides a way to find, structure, and display relationships between concepts which consist of objects with common attributes [35]. Ganter et al. [34] defined a formal concept analysis as follows:

Definition 1: A context is a triple $\mathcal{C} = (\mathcal{O}, \mathcal{A}, \mathcal{R})$ where $\mathcal{O}$ and $\mathcal{A}$ are the set of objects and attributes respectively, and $\mathcal{R} \sqsubseteq \mathcal{O} \times \mathcal{A}$ is a relation among them.

A context is represented as a relation or context table, where rows represent objects and columns represent attributes (see Table 1), with “*x*” indicating that an object has the particular attribute.

Table 1: An example of a context table

$Context\ (\mathcal{O},\mathcal{A},\mathcal{R})$					
		$Attibutes\ \mathcal{A}$			
		female	juvenile	adult	male
$Objects\ \mathcal{O}$	girl	x	x		
	woman	x		x	
	boy		x		x
	man			x	x

Definition 2: A concept $c = (\mathcal{O}, \mathcal{A})$ of a context $(\mathcal{O}, \mathcal{A}, \mathcal{R})$ is part where $O \sqsubseteq \mathcal{O}, A \sqsubseteq \mathcal{A}$, $\alpha(A) = O$ and $\omega(O) = A$. The extent of c is $\pi_o(c) \equiv O$ while the intent is $\pi_\alpha(c) \equiv A$. The set of all concepts of $(\mathcal{O}, \mathcal{A}, \mathcal{R})$ is denoted by $B = (\mathcal{O}, \mathcal{A}, \mathcal{R})$.

A *concept lattice* of a formal context (table) is a collection of all formal concepts (conceptual clusters) equipped with a sub-concept and super-concept hierarchy (Figure 3).

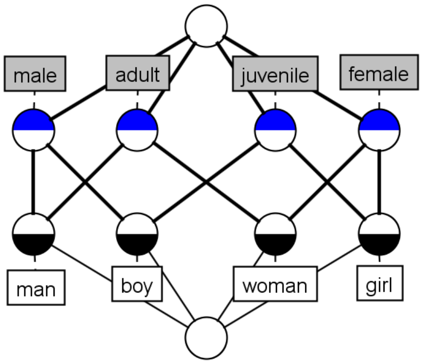


Figure 3: Concept lattice (Galois lattice) of Table 1.

FCA has gained popularity due to its: (1) programming language independence, (2) ability to easily define different views (using different object, attribute combinations), (3) availability of tools to generate context tables and lattices, and (4) analysis being quite inexpensive, especially compared with other dynamic dependency and trace analysis techniques.

FCA applications: Within the SE community, FCA has various applications in SE [35], such as feature interaction, software component retrieval, identifying objects from legacy code, model restructuring, and minimizing test suites. In the knowledge modeling community, FCA has been used for data analysis and knowledge discovery [36] (e.g., knowledge discovery in databases, ontology engineering, and machine learning).

In Chapter 4 we will show how FCA contributes to knowledge model engineering in the literature, and how we adopted this technique in our thesis contribution.

2.4 Vulnerabilities Databases

Security Vulnerability Databases (SVDBs) can be defined as *"a platform aimed at collecting, mailing, and disseminating information about discovered vulnerabilities targeting real computer systems*" [37].

While vulnerability information in existing SVDBs (e.g., National Vulnerability Database - NVD[11]) varies, they typically include a detailed description of reported software vulnerabilities and their potential impacts on existing systems, as well as instructions on how to mitigate these vulnerabilities and countermeasures (such as patches) to thwart further system exploitations. However, security engineers face a major challenge when dealing with existing SVDBs due to the diversity of data models, content, and accessibility of information [38]. For example, many SVDBs use traditional relational databases, while others rely on mailing lists, newsletters, or newsgroups to publish their content.

Existing work on mitigating this problem with heterogeneity has focused on standardizing the publishing of vulnerability information. For example, MITRE[12], a not-for-profit organization that operates research and development centers sponsored by the U.S. federal government, has introduced several standards to be used by the information security community. Three of their most popular standards are: Common Vulnerabilities Exposure (CVE), Common Weakness Enumeration (CWE), and Common Platform Enumeration (CPE). Below, we provide a brief overview of these standards.

[11] https://nvd.nist.gov/

[12] https://www.mitre.org/

CVE[13] is a catalog of publicly known security vulnerabilities and exposures. It is used by SVDBs to simplify the sharing of vulnerability data across different databases and tools by providing a common identifier for vulnerabilities.

CWE[14] is a community-developed dictionary of software weakness types. It provides a standard language of common software security terms in an attempt to remove ambiguity in the use and interpretation of vulnerability information.

CPE[15] is a structural naming scheme for IT systems, platforms, and packages. It provides a standard naming scheme. What CVE and CWE languages both have in common is a need to refer to IT products and platforms in a standardized way that is suitable for machine interpretation and processing.

Another well-known standard, Common Vulnerabilities Scoring System (CVSS[16]), is an open-standard designed to assess the severity of vulnerabilities. This scoring system allows for a standardized ranking of vulnerabilities based on their potential impact.

2.5 Vulnerabilities Detection Techniques

In the SE domain, the concept "*false sense of security*" [39] means that a project manager believes that their system is secure from vulnerability attacks, when in fact that may not be the case. However, involving vulnerability detection techniques (tools) during the software life cycle removes such subjectivity from security vulnerability assessments and gives a project manager quantitative insight into the effectiveness of a projects' security controls.

The discovery and disclosure of new vulnerabilities occurs on a regular basis and is an area of research that is fairly well understood. For example, Liu et. al [40] performed a survey on techniques of discovering vulnerabilities and outlined a number of techniques: static analysis, penetration testing, fuzzing, Vulnerability Discovery Models, and more. These techniques are already deployed in practice while research on new and existing techniques is continuing.

The traditional techniques used to audit software projects against security vulnerabilities are based on static analysis tools (e.g., FindBugs [41]) and vulnerability scanners (e.g., OWASP Dependence-Check [42]). Security static analysis, also called security static code analysis, is a

[13] http://cve.mitre.org
[14] https:cwe.mitre.org
[15] https:cpe.mitre.org
[16] https://www.first.org/cvss/

method of automatic program code debugging without execution. Vulnerability scanner tools play a different role than traditional static analysis tools by scanning the security vulnerability in network or system (such as using vulnerable components in software project). These vulnerability scanners use predefined rules (maintained by security engineers) to identify security violations on networks or systems. In addition, the vulnerability scanner usually uses vulnerability databases (vendor vulnerability database, or third party database such as NVD, SecurityFocus, etc.) to compare the information it finds against known vulnerabilities.

Over the last few years, new versions of vulnerability scanners have taken a different direction in identifying and tracking security vulnerabilities. A study [43] revealed that, in the SE domain, software developers include third party libraries in their applications that contain well-known published vulnerabilities (such as those at NVD) which are hard to detect by traditional static and dynamic analysis tools. However, the gist of this kind of vulnerability scanner is to identify project dependencies and check if there are any known, publicly disclosed, vulnerabilities. These scanners help to validate the inventory of components on the software project. An inventory includes the open-source libraries (third party libraries), projects information (e.g., Maven pom.xml), source code, and other applicable project information. In what follows we give a detailed example of OWASP Dependence-Check [42] (vulnerability scanner tool), which we used to evaluate our proposed approach discussed in Chapters 5 and 6.

OWASP Dependency-Check [42] is a vulnerability scanner tool that identifies software project dependencies and checks if there are any known vulnerabilities. The current version of the tool supports Java and .NET; and additional experimental support has been added for Rube, Node.js, Python, and limited support for C/C++ build systems (autoconf and cmake).

OWASP Dependency-Track is a web application that allows organizations to document the use of third party components across multiple applications and versions. Further, it provides automatic visibility into the use of components with known vulnerabilities.

Dependency-Track embeds the Dependency-Check project which uses public data from the NVD. Dependency-Check uses evidence-based analysis to match vendor, product, and version to the CPE’s identified in CVEs. Dependency-Track v1 is essentially an asset management application for tracking components and their use in each application, along with vulnerabilities.

Dependency-Track/Dependency-Check does not rely on a centralized database of additional data (such as tracking non-disclosed vulnerabilities). Some commercial solutions use these approaches to offer more comprehensive solutions. However, both Dependency-Track and Dependency-Check are open-source projects.

Most of the vulnerabilities scanner tools share the same environment of the vulnerability scanning process. They use what is called **Language Type** and **Evidence Collection** identifications, in order to identify the vulnerable libraries in the third party vulnerability databases.

Language Type. The tools identify the language of the project under scanning by checking what build/package manager is being used. This is done by finding the configuration file for a given build/package manager in the root of the project, or in the location where a configuration file might typically be found. For example, a pom.xml in the root of a project indicates a Maven repository. This information is how the tools distinguish coordinates amongst the various build/package managers.

Evidence Collection. The aforementioned tools collect information about files they scan during the build process, in combination with the language type. The information collected is a set of coordinates called Evidence. There are three types of evidence collected: vendor, product, and version. The evidence for each build/package manager may vary from one to another. For example, the coordinates (evidences) for each language are the following: Java (Maven) uses groupId, artifactId, and version, Node.js (NPM) uses library name and version, Python (PyPi) uses library name and version, and Ruby (Ruby Gems) uses library name and version. However, by sending these evidences, the aforementioned tools are able to check whether the libraries are vulnerable or not by matching against the vulnerability database.

Finally, vulnerabilities scanners that depend on evidence identification may suffer from false positive and false negative results [44].

2.6 Chapter Summary

The work presented in this dissertation is a novel fusion of several techniques from disparate domains. In this chapter, we provided background information to the technologies and concepts used in our research implementations. We will frequently refer to this chapter in subsequent chapters, since the provided information will be extensively required.

In the next chapter, we will survey the SE literatures that use SVDBs in software development, and outline some research opportunities that we are trying to address in this thesis.

Chapter 3

3 A Study on the Use of Vulnerability Databases in Software Engineering

3.1 Introduction

As discussed earlier (in Chapter 1), in response to the constantly increasing number of software vulnerabilities and attacks, various private and public organizations have introduced software vulnerabilities databases (SVDBs) such as the National Vulnerabilities Database (NVD[17]). Each of these databases captures not only different types of vulnerability information, but are also of interest to developers and other stakeholders since they contain valuable information about software flaws, causes of defects, as well as details surrounding how vulnerabilities arise, vulnerability measurements, and guidelines on secure coding practices, etc.[38].

While the information content in SVDBs varies, they typically include a detailed description of each vulnerability and any potential impacts these may have on existing software projects. Additionally, instructions are included on how to mitigate the reported vulnerability, as well as information surrounding countermeasures (such as patches) to thwart further system exploitations and violations. However, working with different SVDBs also involves dealing with heterogeneous data sources since no single repository provides information for all available disclosed software security vulnerabilities [38]. Furthermore, retrieving relevant vulnerability information is not always a straight forward task since every organization and vendor publishing a SVDB relies on proprietary categorization and representation of their vulnerabilities data [38]. While many SVDBs use traditional relational databases to publish their vulnerability

[17] https://nvd.nist.gov/vuln/data-feeds

information, others rely on mailing lists, newsletters, or newsgroups to provide access to these datasets.

Software developers are often unfamiliar with SVDBs and the fact that known vulnerabilities might already be published in these SVDBs. As a result, vulnerabilities affecting their systems (either directly or indirectly through external vulnerable components used by their systems) often remain uncovered. In 2017, the Equifax breach[18], for example, involved a vulnerability in the Apache Struts [45] open source software. The patch came out in SVDBs a couple of months before the breach occurred. Increasing awareness and accessibility of SVDBs, as well as linking vulnerabilities to software artifacts, are an essential aspect in locating vulnerable code, applying existing fixes, and improving the analysis of potential impacts of vulnerabilities.

3.1.1 Definitions

SVDBs can be classified into two major categories, **private** and **public** vulnerabilities databases:

1. **Private** SVDBs are typically managed by for-profit organizations, covering vendor specific product (closed source) vulnerabilities and rarely disclose this information to the public.
2. **Public** SVDBs are organized and maintained typically by non-profit organizations (e.g., Computer Emergency Response Teams (CERT)[19]) and disclose their vulnerability information to the public. These public SVDBs can be further divided into subcategories: *specialized* and *general* (i.e., *common*) SVDBs.
    a. *Specialized* SVDBs are databases covering specific vulnerability aspects (e.g., Mozilla Foundation Security Advisory (MFSA)[20]). Many software manufactures operate their own, highly specialized databases in which publicly known vulnerabilities of their products are documented [46].
    b. *Common* SVDBs are databases that publish vulnerabilities for a number of software projects across vendor boundaries. These databases disclose

[18] https://www.consumer.equifax.ca/canada/equifaxsecurity2017/en_ca/
[19] https://www.cert.org/
[20] https://www.mozilla.org/en-US/security/advisories/

vulnerabilities to the public after being reviewed by security experts (e.g., NVD[21]).

Also, the following definitions help to clarify the meaning of Software Engineering (SE) *tasks* supported by SVDBs, which we used for classifying our literature review.

1. *SE tasks*: any SE method (described in the paper) the SE researchers used to achieve their research goals. For example, SE researchers used empirical research methods to study vulnerability evolution.
2. *SE repository*: a central place to keep software resources which SE researchers can pull from when necessary. SE repositories such as issue tracker systems (e.g., JIRA), version control systems (e.g., GIT), software management repositories (e.g., Maven), etc.

3.1.2 Goals and Outcomes

While the SE community is becoming increasingly aware of these SVDBs, no comprehensive literature survey exists that studies *where* (and *how*) SVDBs are used in SE research. Our literature survey provides insights into current state-of-the-art usage of SVDBs in software development activities. Our primary goals are to characterize and quantify:

- Which SVDBs are commonly used in the SE research community,
- Which SE repositories were mined together with SVDBs,
- Which SE task was being supported through using SVDBs.

To achieve these goals we surveyed 94 articles from the SE literature that used SVDBs to solve SE tasks. Findings from our analysis show that SVDBs are most commonly used for empirical research (e.g., security [empirical] studies), generating security test cases, or modeling vulnerabilities detection techniques. We also found that SE researchers discuss a broad range of security topics, from creating theoretical foundations for new security analysis techniques to implementing them as part of software development environments.

The outcome of our study can be beneficial to different stakeholders since the study provides insights on current trends and open challenges related to the use of SVDBs. In particular, our

[21] https://nvd.nist.gov/vuln/data-feeds

study provides software developers and maintainers the necessary knowledge about the underlying challenges for fixing security vulnerabilities and developing patches by understanding the steps and best practices provided by SVDBs. The study also provides a detailed analysis on how the security vulnerability research landscape has evolved over the past decade. It should be noted that our dataset and results are available online to facilitate the replication and reuse of our findings [47].

3.2 Literature Search and Selection

In this section, we describe the process used to find literature, including our scope, considered venues and search terms, search methodology, and article selection results. We adopt and follow the mapping study approaches suggested by Chen et al. [48], Martin et al. [49], and Petersen et al. [50].

3.2.1 Scope

For this paper we were interested in locating articles that used SVDBs information to solve SE tasks. We focused our attention on articles written between January 2001 and April 2017, more than a full decade of research results. This period was chosen because the SVDBs (e.g., NVD) that propelled vulnerability sharing to become widely adopted were launched in 2005.

Our survey is not a Systematic Literature Review (SLR). Our study aims to define, collect, and curate the disparate literature, arguing and demonstrating that there does, indeed, exist a coherent area of research in the field that can be termed “using SVDBs information for secure Software Engineering”. We hope that this will prove to be an enabling study for future SLRs in this area.

We applied the following *inclusion* criteria:

i. The paper is related to SE, and may have actionable consequences for software users, developers, and maintainers.

ii. The paper is related to SVDBs analysis, concerning the use of collections of vulnerabilities from one or more SVDBs to tackle software issues.

We applied the following *exclusion* criteria:

i. The paper focuses on software vulnerability but does not use vulnerability information from SVDBs.

3.2.2 Considered Venues and Search Terms

Table 2 lists the journals and conference venues that we included in our initial search for articles.

Table 2: The selected journals and conference venues that we considered in our initial article selection process

Type	Acronym	Description
Journal	TSE	IEEE Transactions on Software Engineering
	TOSEM	ACM transactions on Software Engineering & Methodology
	EMSE	Empirical Software Engineering
	JSS	Journal of System and Software
	JCS	Journal of Science of Computer Programming
	SP&E	Software – Practice & Experience
	IST	Information and Software Technology
	JSEP	Journal of Software: Evolution and Process
Conference	ICSE	International Conference on Software Engineering
	ESE/FSE	European Software Engineering Conference / Symposium on the Foundations of Software Engineering
	ASE	International Conference on Automated Software Engineering
	ICSME	International Conference on Software Maintenance
	ICPC	International Conference on Program Comprehension
	ISSRE	International Symposium on Software Reliability Engineering
	ICST	International Conference on Software Testing, Verification and Validation
	ISSTA	International Symposium on Software Testing and Analysis
	ESEM	The ACM / IEEE International Symposium on Empirical Software Engineering and Measurement
	MSR	International Workshop/Working Conference on Mining Software Repositories

Search Terms. We collected the initial set of articles by performing keyword searches on the publisher websites for each of our considered venues. We also searched using aggregate search engines, such as the ACM Digital Library and IEEE Xplore. The keywords and search queries that we used are listed in Table 3.

Table 3: Keyword searches for online library search

Category	Terms
General	("vulnerable" **OR** "vulnerability" **OR** "vulnerabilities" **OR** "vulnerability database" **OR** "vulnerability databases") **AND** ("software engineering")
Domain	("vulnerable" **OR** "vulnerability" **OR** "vulnerabilities" **OR** "vulnerability database" **OR** "vulnerability databases") **AND** ("software requirement" **OR** "software design" **OR** "software coding" **OR** "software testing" **OR** "software verification" **OR** "software evolution" **OR** "software maintenance")

3.2.3 Search Methodology

In order to collect all relevant literature to date that met the scope defined in Section 3.2.1, we performed a systematic search for the terms defined in Table 3, from each repository (defined in Table 2).

Unique papers were collected into a table, and a decision was made based on the inclusion criteria in two stages:

Title and Abstract: We inspected the title and abstract, and removed publications which were clearly irrelevant according to the scope defined in Section 3.2.1.

Body: Results were read fully and a judgement was made on whether the paper (a) meets the key requirements on what is defined as "using SVDBs information for secure Software Engineering" in our scope, or b) is very relevant to the field and so should be included as "expanded literature" to put the main literature into context. Papers matching the requirements of (a) or (b) were included in this survey.

This yielded an initial set of related articles. For each of the articles in the initial set, we considered the citations that were contained in each article for additional relevant articles. Then, we reached our final set of articles.

3.2.4 Snowballing

In addition to the repository searchers specified in Section 3.2.2, we also performed snowballing [51] on many of the included studies. To do this we inspected the studies cited by the study and the publications that subsequently cited the study, using Google Scholar and IEEE Xplore. By performing this process in addition to repository keyword searching, we reduced the risk of omission of relevant literature from this survey.

3.2.5 Article Selection Results

We finally arrived at 94 articles published between 2001 and 2017. Figures 4 and 5 show the distribution of venues and years for the articles. One of the main findings from our first data analysis is that there has been a significant increase in the number of publications (per year) that address vulnerabilities analysis in SE, which is a good indicator for growing research interest in the domain. Secondly, we observe that the main conferences and journals that consider SVDB related articles as a relevant topic are top tier conferences such as ICSE and ASE, and journals such as JSS, IEEE, and TSE.

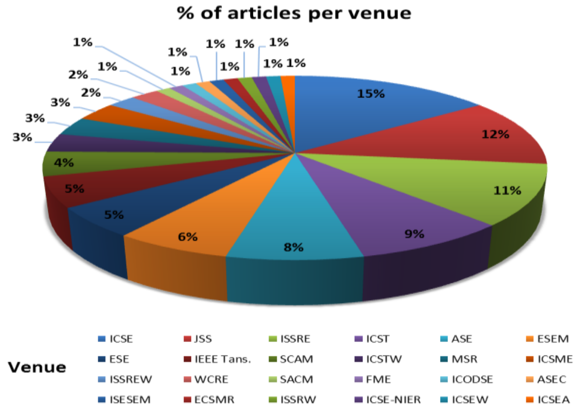


Figure 4: Percent of articles per venue.

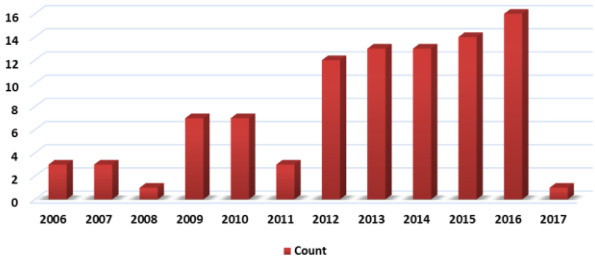


Figure 5: Number of articles per year.

Figure 5 shows that the period between 2001 to 2005 was not included in the final results because papers in that period did not use SVDBs. Figure 4 shows that 15% of ICSE papers deal with SVDBs, while 8% of ASE papers use SVDBs. For the journal papers we found 11% of JSS papers use SVDBs in contrast to 5% of papers in TSE that involved SVDBs in their research.

3.3 Research Trends

In what follows, we report on the major trends which we observed in our literature survey involving the use of SVDBs in the SE lifecycle. In order to structure our literature review, we define a set of attributes that allow us to characterize each of the surveyed articles. Additionally, we define three facets of related attributes, summarized in Table 4.

First and foremost, we are interested in which SVDB was primarily used in the study: *specialized* or *common* SVDBs (defined in Section 3.1.1). Second, we document the SE repository being used together with the SVDB in the article. Finally, we are interested in the SE task (defined in Section 3.1.1) that was being performed along with using SVDBs information. These SE tasks were identified and classified manually from surveyed articles (94 articles). We include a range of tasks to allow a fine-grained view of the literature.

We manually processed each of the articles in our article set and assigned attribute sets to each. The results allow the articles to be summarized and compared along our three chosen facets.

Table 4 shows our three facets: which SVDB was used, which repository was used, and which SE task was being performed. We now analyze the research trends of each facet.

Table 4: The set of attributes we collected on each article

Facet	Attribute	Description
SVDB	Common	Free and open source SVDB which is operated by public organizations that report known vulnerabilities for different software products.
	Specialized	Free and open source SVDB which is operated by private vendors and includes known vulnerabilities specific for those vendors.
Type of paper – SE repository used along with SVDBs	Source code	Involves source code or revision control repository.
	Requirements/design	Involves requirements or design artifacts.
	Logging	Involves execution or search engine logs.
	Bug/vulnerability reports	Involves issue trackers or vulnerability reports from SVDBs.
	Others	Involves Q&A sites, build repositories (e.g., maven, nmp, etc.), emailing list, etc.
Type of paper - SE tasks supported with SVDBs	Empirical research	Papers in this SE task make use of collections of software artifacts (corpora) plus SVDBs from which to derive empirical evidence.
	Source code: Vulnerability analysis	Papers in this SE task perform low level analysis (source code and binary code) to investigate vulnerable code characteristics, develop vulnerability static analysis tools, reverse engineering analysis, etc.
	Security testing	Papers in this SE task introduce vulnerability testing techniques, such as vulnerability test suit generator, model-based testing approaches (using black-and white-box methods), etc.
	Modeling	Papers in this SE task develop models, for example framework models for vulnerability traceability, vulnerability prediction and detection models, knowledge source modeling for interlinking between repositories, etc.
	Risk analysis	Papers in this SE task are more focused on vulnerability assessment and impact during software deployment, for example developing a security assessment approach in incorporating OSS into commercial software systems.
	Other	Any papers that did not fit into the aforementioned classes.

3.3.1 Facet 1: Which SVDBs are Commonly Used in the SE Research Community?

Our survey shows that none of the surveyed SE articles reported on the use of *common* SVDBs prior to 2006 (see Figure 6). This is due to the fact that the first widely-recognized *common* SVDBs (e.g., NVD [38]) became publicly available only in late 2004 and early 2005, with more *specialized* public SVDBs emerging in SE articles in late 2010.

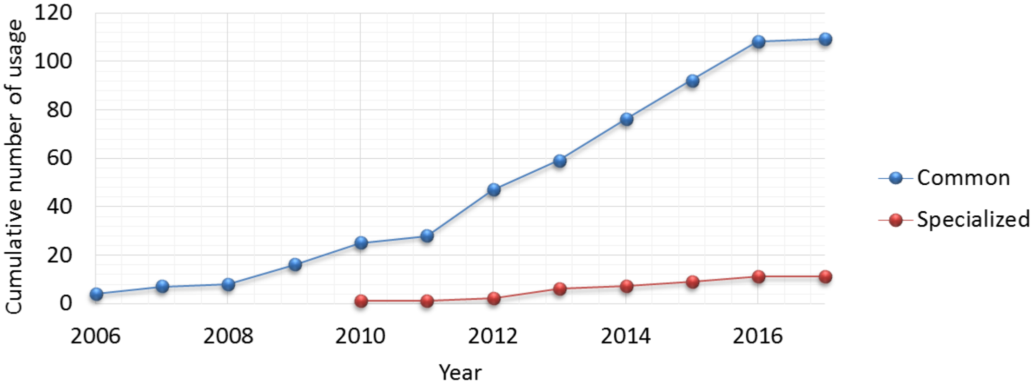


Figure 6: Trends of common vs specialized SVDBs use. The cumulative number of usages indicates the total number of published articles using SVDBs to the year shown on the x-axis.

Our analysis also shows that the majority of surveyed articles (91%) use *common* SVDBs in their work, whereas only 9% rely on *specialized* SVDBs as their primary resource for vulnerability information. Further analysis of *common* SVDBs usage in these papers (see Figure 7) shows that most of the surveyed articles (26%) used NVD as their SVDB of choice, followed by CVE[22] (16%). It should be noted that NVD is based on the CVE dictionary augmented with additional analysis information, a database, and a fine-grained search engine. NVD is synchronized regularly with CVE such that any CVE update will also be reflected in NVD (after approval by the NVD security engineers). NVD includes security checklists, security related software flaws, misconfigurations, affected product names, and impact metrics.

The OWASP is another *common* SVDB which was used by 14% of the surveyed articles. OWASP is dedicated to maintaining a list of Web applications with known security incidents and is commonly used for experiments in testing security vulnerabilities affecting web applications (e.g., [52]–[54]). The Open Source Vulnerability Database (OSVDB), used by 8% of the surveyed articles, is one of the earlier publicly available *common* SVDBs. However, as of April

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

2016 the database is no longer maintained and announced to be shut-down by the vendor. Other popular *common* SVDBs included are CWE (7%) and SecurityFocus (6%). CWE is maintained by MITRE[23] and provides a classification of vulnerabilities types which are commonly used for testing and classifying security attacks. SecurityFocus is an online software systems' security news portal that obtains its data from the Bugtraq[24] mailing list. Bugtraq is an independent source for security vulnerabilities, alerts, and threats.

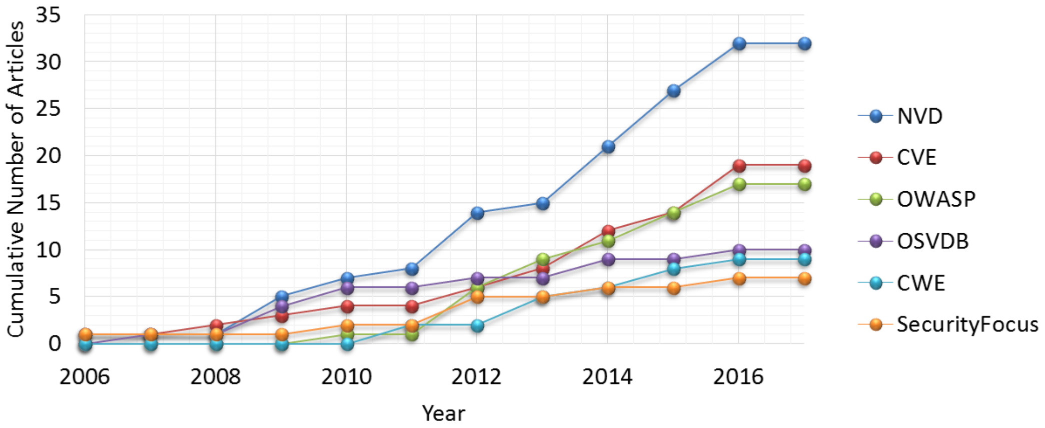


Figure 7: Top 6 *common* SVDBs used in our surveyed articles.

While some articles compare their vulnerability results obtained from one SVDB with results from other SVDBs, only a few studies (e.g., [55], [56]) combined the usage of different *common* SVDBs in their approaches. As shown in [17], combining different *common* SVDBs data sources can increase the zero-day detection performance of vulnerability detection and analysis techniques.

Although the CVE database has been available longer than NVD, most surveyed articles used NVD in their approach. We believe that one of the reasons for the widespread use of NVD is the easy access to vulnerability data through supported feeds and the regular updates to the database. Moreover, we found that little research exists in combining different SVDBs.

[23] https://www.mitre.org/
[24] http://seclists.org/bugtraq/

3.3.2 Facet 2: Which SE Repositories were Mined Together with SVDBs?

Among the most common SE repositories used in combination with SVDBs are source code and bug repositories (see Figure 8). In recent years there has also been increasing research activity in combining SVDBs with logs, requirements, and "*others*" knowledge resources. A main reason for the use of source code and issue tracker repositories in conjunction with SVDBs is that both are typically part of open source systems and SVDBs often contain explicit traceability links to a vulnerability's patch information in either repository. In contrast, requirements and design documents are normally not part of open source systems. We also observed that since 2013 there have been an increasing number of publications combining SVDBs and execution logs. One of the reasons for this is the increasing need to analyze security vulnerabilities that depend on the execution behavior typically found in ultra-large scale or distributed systems (e.g., vulnerability due to a certain load, distributed environments).

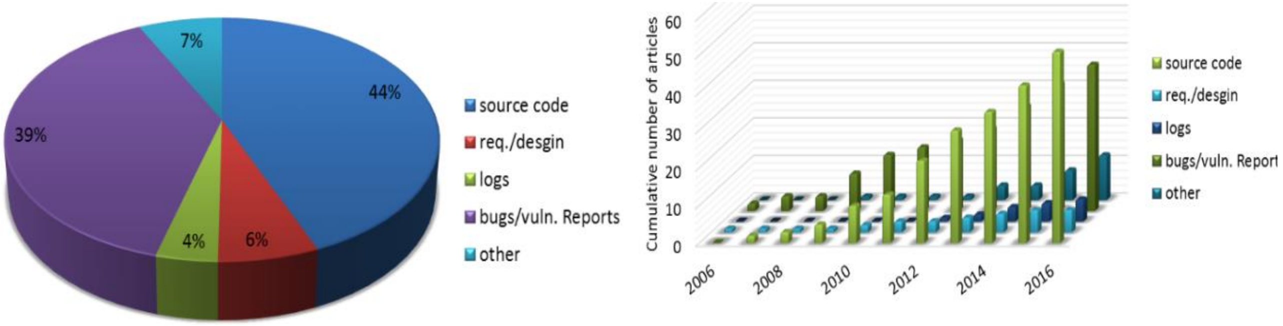


Figure 8: The SE repositories used with SVDBs. On the right, a stacked bar plot which shows the trends on the SE mined repositories. On the left, a plot shows the distribution of the literature survey according to the SE repository used.

The most common SE repositories used and analyzed in combination with SVDBs are source code and issue trackers. Among the main reasons for their widespread use is their availability (e.g., open source projects) and the explicit traceability links among these resources (e.g., commits with cross references to an issue being fixed).

3.3.3 Facet 3: Which SE Task was Being Supported Through Using SVDBs?

The most popular SE tasks in the surveyed articles are empirical research (37% of articles), modeling (20% of articles), source code analysis (for static/dynamic vulnerability analysis 16% of articles), and testing (14% of articles).

Empirical research such as a case study is a task well suited for using SVDBs since the goal of empirical research in SE is to gain knowledge by means of direct and indirect observation or experience [57]. However in order to achieve that, the SVDBs provide empirical evidence (the record of one's direct observations or experiences) of software vulnerabilities which can be analyzed quantitatively or qualitatively.

The modeling task in software security has become popular recently. For example, software threat modeling (e.g., Berger et al. [58]) has been adapted as a key activity in an organization's secure development life cycle (e.g., Microsoft SDL [59]). The modeling task in the surveyed articles include papers discussing vulnerability prediction/detection models (e.g., Chatzipoulidis et al. [60], Scandariato et al. [61], and Shar et al. [62]), or vulnerabilities knowledge source modeling for software ecosystems interrelationships (e.g., Ilo et al. [63], Wu et al. [64], Pham et al. [65], Anbalagan et al. [66], and Cavusoglu et al. [67]). SVDBs play the important role of providing SE researchers structured representations of vulnerability information that affects the security of an application. This helps SE researchers study the vulnerability impacts on the software systems (i.e., how the vulnerability indirectly affects software system components).

Source code analysis includes static/dynamic vulnerability analysis. Our surveyed articles classified in this SE task (i.e., vulnerability static/dynamic approaches) used SVDBs to analyze the system source code for all possible run-time behaviors and seek out coding flaws (e.g., Pasaribu et al. [68], Zheng et al. [69], Møller et al. [70], Shahriar et al. [71], and Wassermann et al. [72]), detecting known vulnerabilities attacks such as SQL injections (e.g., Thome et al. [73]) and buffer-overflow attacks (e.g., Wang et al. [74] and Gao et al. [75]).

For the SE testing task, we find SVDBs as the ideal source for software security testing since many researchers believe that using SVDBs, which contain a wealth of information of software security vulnerabilities, helps to validate the proposed testing techniques with existing known software vulnerability test-cases. The SE testing tasks in the surveyed articles include papers proposing and discussing black-box testing of web-application vulnerabilities (e.g., Ceccato et al. [76]), fuzzing approaches (e.g., Pham et al. [77]), automatic test-cases generation (e.g., Stivalet et al. [78]), etc.

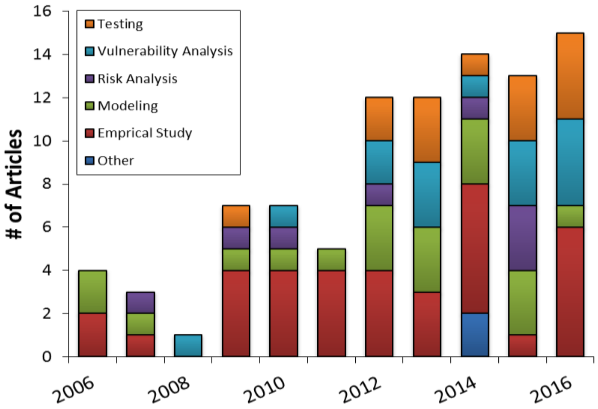


Figure 9: A stacked bar plot which shows the trends of the investigated tasks. The y-axis shows the number of articles published in a given year.

Figure 9 shows a stacked bar plot of the trend of the tasks performed by the surveyed articles across the years reviewed. We see the emergence of articles that conduct studies on collections of vulnerable software systems (risk analysis articles) since 2009. The reason for this may be the increased popularity of *common* SVDBs and their supporting tools, which give researchers the right techniques for analysing vulnerabilities impacts on collections of software systems. In addition, source code vulnerability analysis studies also emerged around 2008, and we noticed several articles that are published on this task each year since 2008.

The tasks in the *"other"* category include benchmarking methodology for the security of software-based systems (e.g., Mendes et al. [56]) and techniques for counteracting web browser exploits (Min et al. [79]).

Most research that uses SVDBs information performs empirical study, testing, or modeling.

3.4 Common Uses of SVDBs on Software Engineering Tasks

In this section we describe and evaluate in detail the surveyed articles that used SVDBs along with SE repositories to perform some SE tasks. We organize the work into subsections by SE task (as is described in Table 4). We provide a brief description of each task, followed by a presentation of the relevant articles.

3.4.1 Empirical Research

In SE, empirical research is a validating process that compares what the researchers believe to what they observe [57]. Specifically, empirical research helps the researchers understand how things (in software systems) work and allows researchers to use this understanding to materially alter their world. Empirical research takes many forms. It is realized not only as formal experiments, but also as case studies, surveys, and prototyping exercises as well.

In our survey, among the SE tasks which are supported by SVDBs, empirical research is the most common. The results are shown in Table 5, which shows summaries of the articles classified in this SE task, including the reference of each article, publication year, name of the used SVDB, what the SVDB was used for, and self-classification.

In this SE task, we manually classified the articles under this category based on *self-classification* criteria. Note that we do not claim to have surveyed such empirical research studies in SE research comprehensively as this is not the focus of this study, but it can be a direction for future work. However, we tried to understand and classify the empirical research based on *self-classification* criteria (as it is suggested by [80]). For each paper we manually captured what words authors used to describe their efforts (e.g., case study, experiment). We collected explicit *self-classification* from the author's keywords mentioned in the paper or we looked for sentences in the paper such as "we have conducted a case study" and concluded that the current paper is empirical research and, more specifically, a case study. Some of the *self-classifications* were very detailed and not precise (e.g., "empirical analysis," "empirical assessment," or "experiment study"); in such cases we reduced the type to a simpler version (e.g., an empirical study). As a result, we came up with four sub-categories of the empirical research area: case study, exploratory, comparison, and empirical study. As seen from Table 5, authors most often used terms such as “exploratory” and “empirical study” to describe their research. In some cases, papers contained more than one *self-classification*.

Table 5: Results of SE empirical research articles using SVDBs

Reference	Year	SVDBs	How SVDBs Used	Self-classification
Hafiz et al. [81]	2016	SecurityFocus	Extracted vulnerabilities’ authors’ information.	Empirical study (survey)
Munaiah et al. [82]	2016	NVD	Downloaded Chromium project’s vulnerabilities.	Case study (Chromium project)
Ye et al. [83]	2016	CVE	Extracted 100 buffer overflow vulnerabilities.	Empirical study (buffer overflow attack)
di Biase et al. [84]	2016	CVE	Downloaded Chromium project’s vulnerabilities reports.	Case study (modern code review)
Jimenez et al. [85]	2016	NVD	Downloaded vulnerabilities for Linux kernel.	Case study (Linux Kernel)
Murtaza et al. [86]	2016	NVD	Mining NVD.	Exploratory study (mining)
Camilo et al. [87]	2015	NVD	Related to Munaiah et al. [82]	
Fang et al. [88]	2014	SecurityFocus	Related to Hafiz et al. [81], but this paper focused on how buffer overflow attack types.	
Tan et al. [89]	2014	NVD	The authors used NVD to find the security bugs for their analyzed studied systems.	Empirical study (bug characteristics)
Walden et al. [55]	2014	NVD and proprietary vulnerabilities Datasources	Mining NVD.	Comparison study
Stuckman et al. [90]	2014	CVE	The authors used CVE-IDs for traceability.	Exploratory study (mining)
Massacci et al. [91]	2014	NVD and OSVDB	Extracted vulnerability information.	Comparison study
Wijayasekara et al. [92]	2014	CVE and Bugzilla	Downloaded vulnerability information for vulnerable releases of Linux kernel.	Exploratory study (mining bugs vs vulnerabilities)
Meneely et al. [93]	2013	NVD	The authors used NVD to trace 68 vulnerabilities using CVE-IDs.	Exploratory study (vulnerability-contributing commits)
Meneely et al. [94]	2013	CWE	Studied types of vulnerability attacks.	Empirical study (survey)
Lee et al. [95]	2013	CVE	Downloaded CVE vulnerabilities for Fedora open source.	Case study (RedHat Fedora)
Shahzad et al. [96]	2012	NVD, OSVDB, and FVDB	The authors aggregated 46310 vulnerabilities SVDBs.	Exploratory study (vulnerability life cycle)
Goseva-Popstojanova et al. [97]	2012	NVD, OWASP, and SecurityFocus	Used for annotating specific vulnerability classes of attacker activities.	Empirical study (classifying web systems vulnerabilities)
Liu et al. [98]	2012	NVD	Downloaded 11, 395 vulnerability entries and related information.	Empirical study (vulnerability prioritization)

Wijayasekara et al. [99]	2012	CVE	Related to Wijayasekara et al. [92]	
Austin et al. [100]	2011	CWE	The authors studied the type of vulnerabilities.	Comparison study (two electronic health record systems)
Smith et al. [101]	2011	CWE	The authors used SVDB as a source for vulnerability classifications.	Empirical study (prioritizing security V&V)
Zhang et al. [102]	2011	NVD	Mining NVD.	Empirical study (mining NVD data)
Zaman et al. [103]	2011	Mozilla Foundation Security Advisory (MFSA)	Extracting and studying vulnerabilities reports.	Case study (Firefox vulnerability vs. bugs)
Huynh et al. [104]	2010	OSVDB and SecurityFocus	Studying whether the vulnerabilities share any common properties or not.	Empirical study (web app vulnerability characteristics)
Zimmermann et al. [105]	2010	NVD	The authors extracted vulnerabilities entries for Windows Vista.	Empirical study (evaluate the efficacy of pre-defined metrics predicting vulnerabilities on Windows Vista)
Neuhaus et al. [106]	2010	NVD	Mining NVD.	Exploratory study (vulnerability trends)
Mauczka et al. [107]	2010	FreeBSD adivsory and OWASP	Studying the security vulnerabilities behaviour.	Exploratory study (mining)
Wal et al.[108]	2009	NVD and CVE	Extracting vulnerabilities counts for web applications.	Empirical study (vulnerability characteristics in PHP applications)
Anbalagan et al. [109]	2009	NVD	Trace open source projects issue trackers to their security vulnerabilities.	Exploratory study (tracing vulnerability for Fedora, Ubuntu, and OpenSuse)
Anba et al. [110]	2009	NVD	Extracting vulnerabilities data for Linux distributions.	Empirical study (vulnerability reports characteristics of Linux Distributions)
Vache [111]	2009	OSVDB	Downloaded vulnerability information to study the vulnerability life cycle and the exploit appearance.	Exploratory study (characterized quantitatively the vulnerability life cycle and the exploit appearance)
Telang et al. [112]	2007	CERT	The authors extracted vulnerabilities information.	Empirical study (vulnerability impact on product market price)
Alhazmi et al. [113]	2006	NVD and Netcraft	The authors extracted vulnerabilities information.	Comparison study
Frei et al. [114]	2006	OSVDB, NVD	The authors downloaded vulnerability reports.	Exploratory study (understanding of the vulnerability lifecycle)

Case Studies. This type of empirical research uses specific open-source projects to study and analyze the security vulnerability. For example, Munaiah et al. [82] performed an in-depth analysis of the Chromium project to empirically examine the relationship between bugs and vulnerabilities. They used NVD as a source of the Chromium vulnerabilities. The bug reports were collected from Google code portal as a source for the Chromium bugs. Towards the same goal, Zaman et al. [103] used Firefox as a case study to investigate how different types of bugs (performance and security bugs) differ from each other and from the rest of the bugs in a software project. These researchers used Mozilla Foundation Security Advisory (MFSA) (i.e., *specialized* SVDB).

The main objective of the study by Jimenez et al. [85] was to determine the effectiveness of vulnerability prediction models (e.g., Software Metrics, and Text Mining) to distinguish between vulnerable and non-vulnerable software components of the Linux Kernel under different scenarios. The case study dataset was based on extracting vulnerabilities reports from the NVD and the bug/commit-reports from Linux Kernel repository (Bugzilla).

Lee et al. [95] verified the assumption about using software reliability models for security assessment. They investigated a range of Fedora open source software security problems to see if some of the basic assumptions behind software reliability growth models hold for discovery of security problems in non-attack situations. They used CVE identifications to locate security discussions from open source RedHat Bugzilla data[25] for Fedora.

di Biase et al. [84] conducted an empirical case study aimed to fill the gap between Modern Code Review (MCR) and post-release bugs (software quality). They explored the MCR process in the Chromium open source project. They used the CVE dataset to correlate the vulnerability (security) issues of the project to the reported ones in the project's host portal.

Comparison Studies. We have found comparisons in 4 papers. The comparison is made for the technique, approach, or tool that was introduced in the study—to evaluate the proposed technique, approach, or tool. However, in this SE task, we find that all 4 papers (Massacci et al. [91], Walden et al. [55], Alhazmi et al. [113], and Austin et al. [100]) share the same goal of performing a comparison study to validate and evaluate vulnerability prediction/discovery models. For example, Massacci et al. [91] introduced an empirical comparison methodology to evaluate vulnerability discovery models (e.g., Alhazmi and Malaiya Logistic model [115], and Logarithmic Poisson [116]), and they evaluated the performance of VDMs with two dimensions (quality and predictability). The vulnerability data used in the study were extracted from two sources: (1) *common* SVDBs (e.g., NVD, OSVDB, etc.), and (2) *specialized* SVDBs, the

[25] https://bugzilla.redhat.com/

vulnerability database maintained by the software vendor (e.g., MFSA, Microsoft Security Bulletin). In some cases, authors used open source and commercial products (e.g., Apache[26] and Microsoft IIS [27] HTTP servers) for measuring and enhancing prediction capabilities of vulnerability discovery models as described in [113].

Exploratory Studies. In this SE task we noticed exploratory research articles mainly focused on studying vulnerability evolution (including vulnerabilities trends, patterns, and life cycle). The studies involved more understating of the nature of vulnerabilities databases by extracting the vulnerabilities features, and involved mining techniques. We found studies by Murtaza et al. [86] and Neuhaus et al. [106] in which they mined the vulnerabilities data extracted from NVD to analyze the security vulnerabilities' trends and patterns. Anbalagan et al. [109], Stuckman et al. [90] and Meneely et al. [93] mined SVDBs (e.g., CVE and NVD) to trace security vulnerabilities. For example, Meneely et al. [93] conducted an exploratory study to explore the properties of vulnerability-contributing commits in order to check when a software vulnerabilities patches go bad. In this paper, the authors traced 68 vulnerabilities (extracted from NVD) in the Apache HTTP server back to the version control commits that contributed to the vulnerable code originally.

Other exploratory studies involved large scale exploratory analysis of software vulnerability life cycles (e.g., Shahzad et al. [96], Vache et al. [111], and Frei et al. [114]) and attempted to understand the security behaviour of certain projects (e.g., [107]). Mauczka et al. [107] used mining techniques to extract security changes in FreeBSD[28]. They used the gathered security changes to find out more about the nature of security in the FreeBSD project, and they tried to establish a link between the identified security changes and a tracker for security issues (security advisories). For their study the authors used vulnerability information provided by OWASP and CWE repositories.

Empirical Studies. The rest of the empirical research articles in our literature survey were classified as empirical studies. Our further manual analysis of the related articles of these empirical studies shows that: empirical studies used SVDBs for surveying vulnerability reporters

[26] https://httpd.apache.org/
[27] https://www.iis.net/
[28] https://www.freebsd.org/

[81] and studying the security knowledge of SE students [94]. For example, Hafiz et al. [81] conducted an empirical study on 127 vulnerability reporters to understand the methods and tools used during the discovery of the software vulnerability and whether the community of developers exploring security vulnerability differ in their approach from another community of developers exploring a different vulnerability. The study was based on vulnerability reporters in the SecurityFocus repository. In addition, the authors analyzed certain types of vulnerability attack reports (e.g., SQL injection and cross site scripting vulnerabilities) extracted from the SecurityFocus repository. Similarly to [81], Fang et al. [88] replicated the same experiment but with a focus only on how buffer overflow attack types are discovered.

Some studies were concerned with comprehending the security bugs' (vulnerabilities) characteristics (e.g., Tan et al. [89], Huynh et al. [104], Wal et al. [108], Anba et al. [110], Ye et al. [83], Wijayasekara et al. [92], and Wijayasekara et al. [99]). For example, Tan et al. [89] studied the bug characteristics in open-source software (e.g., the Linux kernel, Mozilla, and Apache). They used bug related vulnerabilities (2,060 vulnerability reports) from the NVD repository and manually studied these bugs in three dimensions—root causes, impacts, and components. Some studies focused on analyzing vulnerabilities' characteristics for web applications, as discussed in papers [108] and [104]. Some papers studied vulnerabilities characteristics that affect certain development platforms such as Linux environment [110]. Also, some studies investigated common vulnerabilities attacks. Ye et al. [83] performed a quantitative and qualitative study on static buffer overflow detection. The data was collected from buggy and fixed versions of 100 buffer overflow bugs from 63 real-world projects based on the vulnerability reports from the CVE dataset.

Wijayasekara et al. [92] and [99] mined bug databases for unidentified software vulnerabilities. The authors claimed that it has been suggested in previous work that some bugs are only identified as vulnerabilities long after the bug has been made public. These vulnerabilities are known as hidden impact vulnerabilities. Wijayasekara et al. presented a vulnerability analysis from January 2006 to April 2011 for two well-known software packages: Linux kernel and MySQL. They showed that 32% (Linux) and 62% (MySQL) of vulnerabilities discovered in this time period were hidden impact vulnerabilities. The study also showed that the percentage of hidden impact vulnerabilities has increased from 25% to 36% in Linux and from

59% to 65% in MySQL in the last two years. They used CVE database as the source of vulnerabilities for the studied subjects.

Liu et al. [98] presented an empirical study to improve vulnerability rating and scoring system (VRSS-based) vulnerability prioritization using an analytic hierarchy process. They analyzed 11, 395 CVE vulnerabilities to expose the differences among three current vulnerability evaluation systems (IBM Internet Security Systems (ISS) X-Force [117] , Common Vulnerability Scoring System (CVSS) [118], and Vulnerability Rating Systems (VRSS) [119]).

Papers introduced by Goseva-Popstojanova et al. [97] and Smith et al. [101] used machine learning algorithms to classify software vulnerabilities. For example, Goseva-Popstojanova et al. [97] used multiclass machine learning methods to classify malicious behaviors aimed at web systems—Web 2.0 applications (i.e., a blog and wiki). They used the vulnerability data from OWASP, NVD, and SecurityFocus. Smith et al. [101] showed a classification method of detecting different types of web application vulnerabilities (subject systems are WordPress, a blogging application, and WikkaWiki, a wiki management engine). The authors relied on the vulnerabilities types extracted from the CWE database.

Last, we found empirical studies that used SVDBs information in investigating vulnerabilities impacts on the software stock market (e.g., [112]). In their paper, Telang et al. [112] completed an empirical analysis of the impact of software vulnerability announcements on firm stock price. They collected data from leading national newspapers and industry sources, such as the Computer Emergency Response Team (CERT), by searching for reports on published software vulnerabilities. The researchers showed that vulnerability announcements lead to a negative and significant change in a software vendor's market value.

The common use of SVDBs in this SE task is downloading/extracting vulnerability information for the subject systems. In some cases, exploratory studies were more involved in comprehending the vulnerability behaviour (e.g., trends and life cycle) by extracting the vulnerability features provided by SVDBs.

3.4.2 Modeling

Modeling in SE research is primarily concerned with reducing the gap between software problems and implementation through the use of models that describe complex systems at multiple levels of abstraction and from a variety of perspectives [120]. Table 6 summarizes the surveyed articles classified in this SE task, showing title of each article, publication year, name of the used SVDB, and what the SVDB was used for.

Table 6: SE articles using SVDBs in SE modeling task

Reference	**Title of Paper**	**Year**	**SVDBs**	**How SVDBs Used**
Alqahtani et al. [44]	SV-AF - A Security Vulnerability Analysis Framework	2016	NVD	Modeling vulnerability information.
Morrison et al. [121]	A Security Practices Evaluation Framework	2015	OWASP and NVD	Modeling secure coding practices knowledge.
Ilo et al. [63]	Combining Software Interrelationship Data Across Heterogeneous Software Repositories	2015	NVD	Modeling vulnerability information.
Chatzipoulidis et al. [60]	Information Infrastructure Risk Prediction Through Platform Vulnerability Analysis	2015	NVD	Used vulnerability measures information.
Scandariato et al. [61]	Predicting Vulnerable Software Components via Text Mining	2014	NVD	Extracted seven vulnerability reports related to Android apps.
Murtaza et al. [122]	Total ADS: Automated Software Anomaly Detection System	2014	CVE	Downloaded specific vulnerabilities examples.
Milenkoski et al. [123]	Experience Report: An Analysis of Hypercall Handler Vulnerabilities	2014	CVE	Downloaded 23 vulnerabilities.
Shar et al. [62]	Mining SQL Injection and Cross Site Scripting Vulnerabilities Using Hybrid Program Analysis	2013	CVE and PMASA	Downloaded vulnerability information.
Berger et al. [58]	Extracting and Analyzing the Implemented Security Architecture of Business Applications	2013	CWE	Extracted vulnerability attacks examples.
Almorsy et al. [124]	Automated Software Architecture Security Risk Analysis Using Formalized Signatures	2013	CAPEC	Applying predefined security pattern attacks.
Shar et al. [125]	Predicting Common Web Application Vulnerabilities from Input Validation and Sanitization Code Patterns	2012	SecurityFocus	Downloaded vulnerable projects.
Almorsy et al. [126]	Supporting Automated Vulnerability Analysis Using Formalized Vulnerability Signatures	2012	OWASP	Downloaded vulnerability information and attack types.
Gauthier et al. [127]	Fast Detection of Access Control Vulnerabilities in PHP Applications	2012	OWASP and CVE	Downloaded specific number of vulnerability information.

Wu et al. [64]	Empirical Results on the Study of Software Vulnerabilities (NIER Track)	2011	CWE, CVE	Modeling vulnerability attack and relations with software projects.
Pham et al. [65]	Detecting Recurring and Similar Software Vulnerabilities	2010	CERT and CVE	Downloaded and analyzed 2, 598 vulnerabilities.
Anbalagan et al. [66]	Towards a Unifying Approach in Understanding Security Problems	2009	OSVDB, NVD	Downloaded and analyzed 43, 710 vulnerabilities.
Cavusoglu et al. [67]	Efficiency of Vulnerability Disclosure Mechanisms to Disseminate Vulnerability Knowledge	2007	CERT	Downloaded and studied the disclosure policy of vulnerabilities reports.
Xu et al. [128]	Threat-Driven Modeling and Verification of Secure Software Using Aspect-Oriented Petri Nets	2006	SecurityFocus	Studied specific vulnerability attacks.
Byers et al. [129]	Modeling Software Vulnerabilities With Vulnerability Cause Graphs	2006	CVE	Used vulnerability information as a case study.

Table 6 shows articles classified in this SE task. We manually classified these articles based on their content into four categories: knowledge modeling papers, vulnerability prediction models papers, threat modeling papers, and others. In what follows, we provide a summary of each article in each category.

Knowledge modeling. In our survey, this SE task (i.e., modeling) often uses SVDBs as a source of knowledge modeling for vulnerability analysis and interlinking (e.g., Alqahtani et al. [44], Morrison et al. [121], Ilo et al. [63], Wu et al. [64], Pham et al. [65], Anbalagan et al. [66], and Cavusoglu et al. [67]). Alqahtani et al. [44] introduced a vulnerability analysis framework based on Semantic Web technologies, and used NVD as an example of SVDBs that can be integrated with SE knowledge sources (e.g., source code repositories, issue tracker systems) to study software security evolution. In other related work, Ilo et al. [63] introduced abstract research, which proposed an ontology for the semantic modeling of the relationships between SE ecosystems as linked data. They also proposed to evaluate their approach by integrating the data of several ecosystems (Maven[29] and NVD) and demonstrated its usefulness by creating tools for vulnerability notification and license violation detection. Wu et al. [64] proposed organizing the information in project repositories around *semantic templates* (semantic templates are generalized patterns of relationship between software elements and faults, and their association with known higher level phenomena in the security domain [130]). In this paper the authors presented preliminary results of an experiment conducted to evaluate the effectiveness of using

[29] https://search.maven.org/

semantic templates as an aid to studying software vulnerabilities. In the experiment they used several reported vulnerabilities in the Apache Web Server from NVD.

We found SVDBs used as a knowledge source for software security practices—for example, Morrison et al. [121] who proposed a security practices evaluation framework. The goal of their framework is to aid software practitioners in evaluating security practice use in the development process by defining and validating a measurement framework for software development security practice use and outcomes. For evaluating the framework they proposed the use of historical data and industrial projects from different repositories such as OWASP and NVD.

By using vulnerabilities from OSVDB and NVD, Anbalagan et al. [66] discussed a model that captures software systems vulnerabilities status including the type of vulnerabilities, their disclosure status, exploit status, and their correction status. They mapped vulnerabilities for Bugzilla and FEDORA products to the model, with the goal to estimate model parameters in terms of studying the relationship between security problems and security exploits. In related work, Pham et al. [65] showed an approach to detecting recurring and similar software vulnerabilities. They proposed SecureSync, an automatic approach to detect and provide suggested resolutions for recurring software vulnerabilities on multiple systems sharing/using similar code or API libraries. They extracted and analyzed vulnerabilities reports from the US-CERT database. Additionally, the authors used some vulnerability information from the CVE database.

Cavusoglu et al. [67] studied how vulnerabilities should be disclosed to minimize the associated social loss. The authors developed a game-theoretical model [67] in which the coordinator minimizes the societal loss, which includes both damage to vulnerable firms and the patch development cost to the software vendor. The proposed model consists of four stakeholders in the vulnerability knowledge dissemination process: software developer (vendor), software deployers (firms), vulnerability identifier (benign user or hacker), and central coordinator (CERT[30]).

Vulnerability Prediction Models (VPM). We also found that the surveyed articles in this SE task used SVDBs for VPMs (e.g., Chatzipoulidis et al. [60], Scandariato et al. [61], Shar et al. [62], and Shar et al. [125]). VPM is a field of study which aims at automatically classifying software

[30] Computer Emergency Response Team (CERT): https://www.us-cert.gov/

entities as vulnerable or not. For example, Shar et al. [62] and [125] showed a machine learning technique to predict common web application's vulnerabilities from input validation and sanitization code patterns. They proposed a set of static code attributes that represent the characteristics of input validation and sanitization routines for predicting the two most common web application vulnerabilities—SQL injection and cross site scripting. The test subjects' vulnerability information was extracted from the SecurityFocus repository and CVE dataset. In related work, Scandariato et al. [61] presented an approach based on machine learning to predict which components of a software application contain security vulnerabilities. The approach is based on text mining the source code of the components.

Chatzipoulidis et al. [60] aimed to provide a complementary approach to existing vulnerability prediction solutions and launched the measurement of zero-day risk by introducing a risk prediction methodology for an information infrastructure. The practicality of the risk prediction methodology is demonstrated with an implementation example from the electronic banking sector and vulnerability information extracted from NVD.

Threat Modeling. Some of the surveyed articles in this SE task introduced threat modeling approaches (e.g., Berger et al. [58], Xu et al. [128]) using SVDBs as a knowledge source. Threat modeling is an approach for analyzing the security of an application [131]. It is a structured approach that enables you to identify, quantify, and address the security risks associated with an application [131]. Berger et al. [58] proposed a technique that automatically extracts the implemented security architecture of Java-based business applications from the source code. They carried out threat modeling on this extracted architecture to detect security flaws. To create the proposed approach's knowledge base containing well-known threats as well as possible mitigations, the researchers inspected the CWE database which lists a number of security problems and their consequences, as well as potential mitigations. Xu et al. [128] presented a formal threat-driven model approach which explores explicit behaviors of security threats as the mediator between security goals and applications of security features. They demonstrated their approach through a systematic case study on the threat-driven modeling and verification of a real-world shopping cart application (using vulnerability information from SecurityFocus database).

Others. Last, we find other papers in this SE task that introduce modeling approaches with help from SVDBs information. For example the use of SVDBs in modeling specific vulnerabilities for anomaly detection (e.g., Murtaza et al. [122]), using SVDBs information to model and understand hypercall[31] handler vulnerabilities (e.g., Milenkoski et al. [123]), defining and modeling formal vulnerabilities signatures by using vulnerabilities signatures provided by SVDBs (e.g., Almorsy et al. [124], Almorsy et al. [126]), and locating the vulnerability root cause by modeling SVDBs vulnerabilities based on the vulnerability cause graph technique (e.g., Byers et al. [129]).

From our surveyed papers, we find that the common use of SVDBs in this SE task manifold in three reasons: extract vulnerabilities information from SVDBs to improve vulnerability prediction/detection models, to trace and localize vulnerability in software systems, and to enrich proposed frameworks for software security practices.

3.4.3 Source Code: Vulnerability Analysis

Source code vulnerability analysis is a static/dynamic analysis method destined to analyze the source code and/or compiled versions of code to help find security flaws [132]. Table 7 summarizes the articles classified in this SE task, showing the title of each article, publication year, name of the used SVDB, and what the SVDB was used for.

Table 7: SE articles using SVDBs in vulnerability analysis

Reference	Title of Paper	Year	SVDBs	How SVDBs Used
Wang et al. [74]	deExploit: Identifying Misuses of Input Data to Diagnose Memory-Corruption Exploits at the Binary Level	2017	Exploit-DB	Downloaded selected exploit codes.
Sampaio et al. [52]	Exploring Context-Sensitive Data Flow Analysis for Early Vulnerability Detection	2016	OWASP	Investigated vulnerability source code and security attack examples.
Nguyen et al. [133]	An Automatic Method for Assessing the Versions Affected by a Vulnerability	2016	NVD	The authors downloaded vulnerable releases.

[31] Hypercalls are software traps (i.e., interrupts) from a kernel of a fully or partially para-virtualized guest Virtual Machine (VM) to the hypervisor.

Gao et al. [75]	BovInspector: Automatic Inspection and Repair of Buffer Overflow Vulnerabilities	2016	Selected Vulnerable Projects	Selected vulnerable releases, and used CVE information as motivation example in the introduction and related work.
Ming et al. [134]	StraightTaint: Decoupled Offline Symbolic Taint Analysis	2016	CVE	The authors extracted specific vulnerabilities from CVE .
Thome et al. [73]	Security Slicing for Auditing XML, XPath, and SQL Injection Vulnerabilities	2015	OWASP	Downloaded vulnerable web applications.
Theisen et al. [135]	Approximating Attack Surfaces with Stack Traces	2015	NVD	Downloaded vulnerable releases of Windows.
Renatus et al. [136]	Improving Prioritization of Software Weaknesses Using Security Models with AVUS	2015	CWE and CVSS	Used for rating and classification.
Pasaribu et al. [68]	Input Injection Detection in Java Code	2014	CVE and OWASP	Downloaded vulnerable web applications.
Zheng et al. [69]	Path Sensitive Static Analysis of Web Applications for Remote Code Execution Vulnerability Detection	2013	Selected Vulnerable Projects	The authors used specialized vulnerable web apps that suffer from remote code execution.
Coker et al. [137]	Program Transformations to Fix C Integers	2013	SRD and CWE	Downloaded vulnerable applications and vulnerability information.
Ofuonye et al. [138]	Securing Web-Clients with Instrumented Code and Dynamic Runtime Monitoring	2013	MS bulletins	Use one vulnerability example for case study.
Møller et al. [70]	Automated Detection of Client-State Manipulation Vulnerabilities	2012	OWASP	The authors downloaded vulnerable web applications and vulnerability information.
Bernat et al. [139]	Structured Binary Editing with a CFG Transformation Algebra	2012	NVD	The authors downloaded CVEs for Apache subject system.
Shahriar et al. [71]	Client-Side Detection of Cross-Site Request Forgery Attacks	2010	OSVDB	Downloaded vulnerable projects.
Wassermann et al. [72]	Static Detection of Cross-Site Scripting Vulnerabilities	2008	CVE	The authors used CVE vulnerability examples.

Our survey shows a focus of research activity related to static (and dynamic) analysis research. Static analysis is performed in a non-runtime environment [140]. Typically static analysis research will focus on inspecting program code for all possible run-time behaviors and seek out coding flaws, back doors, and potentially malicious code. Dynamic analysis adopts the opposite approach and is executed while a program is in operation [140]. SVDBs in this context are often used as a source of validating the proposed approaches with known vulnerability attacks, understanding specific types of security attacks, tracking and evaluating the attacks patterns, and checking and investigating the root cause of the attack in the source code.

We found that articles classified in this SE task are dedicated to static analysis to detect and analyze known security vulnerabilities (e.g., Thome et al. [73], Pasaribu et al. [68], Zheng et al. [69], Møller et al. [70], Shahriar et al. [71], Wassermann et al. [72] and Coker et al. [137]). For example, Thome et al. [73] introduced an approach to assist security auditors by defining and experimenting with pruning techniques to reduce original program slices to what they refer to as security slices. The approach focused on extracting relevant security vulnerabilities implemented in web application source code such as XMLi [141], XPathi [142], and SQLi vulnerabilities. The authors validated their approach on vulnerable web applications downloaded from the OWASP repository. Pasaribu et al. [68] introduced a tool for input injection (SQL injection, command injection, and cross-site scripting) detection in java code. The authors extended an existing static analysis tool—FindBugs[32]—for input injection detection. The tool was verified on vulnerable web applications, WebGoat from OWASP and ADempiere[33] vulnerable version from CVE. Zheng et al. [69] used a static analysis approach to detect Remote Code Execution (RCE) attacks in web apps. They proposed a path- and context-sensitive interprocedural analysis to detect RCE vulnerabilities. The authors used selected vulnerable web apps that suffer from remote code execution. Shahriar et al. [71] and Wassermann et al. [72] used static analysis techniques to detect Cross-Site Request Forgery (CSRF) and Cross-Site Scripting (XSS) vulnerabilities attacks in web applications, respectively. Shahriar et al.'s [71] approach relies on the matching of parameters and values present in a suspected request with a form's input fields and values that are displayed on a webpage. They validated their approach on web applications that have been reported to contain CSRF in OSVDB. Wassermann et al. [72] presented a static analysis for finding XSS vulnerabilities that directly addresses weak or absent input validation. The proposed approach combines work on tainted information flow with string analysis. The approach is evaluated in web applications and finds both known and unknown vulnerabilities using vulnerabilities information extracted from CVE database.

Møller et al. [70] introduced an approach for automated detection of client-state manipulation vulnerabilities in web applications. They presented a static analysis for frameworks such as Java Servlets, JSP, and Apache Struts. Given a web application archive as input, the analysis identifies occurrences of client state and infers the information flow between the client state and the shared application state on the server. To validate their proposed approach, the researchers ran experiments on a collection of open source web applications, some of them introduced as vulnerable apps from OWASP.

[32] http://findbugs.sourceforge.net/

[33] http://adempiere.org/site/

Coker et al. [137] discussed static program transformations to fix C integers. The paper describes three program transformations that fix C integer problems—one explicitly introduces casts to disambiguate type mismatch, another adds runtime checks to arithmetic operations, and the third one changes the type of a wrongly-declared integer. The authors validated their proposed approach on dataset from NIST's SAMATE reference dataset[34] and CWE information.

Dynamic analysis research on monitoring web applications security is discussed in [138] and [139]. Ofuonye et al. [138] introduced an approach to securing web-clients with instrumented code and dynamic runtime monitoring. The proposed approach seeks to isolate exploitable security vulnerabilities and enforce runtime policies against malicious code constructs. To validate their approach the authors used four case studies, and for one of them they used a publicly available proprietary vulnerability database (Microsoft bulletins). Bernat et al. [139] introduced a dynamic approach for structured binary editing with control flow graph (CFG) transformation algebra. They defined an algebra of CFG transformations that is closed under a CFG validity constraint, thus ensuring that users can arbitrarily compose these transformations while preserving structural validity. They demonstrated the usefulness of their approach by creating a patching tool that closes three vulnerabilities (extracted from NVD repository) in a running Apache HTTPD server without interrupting the server's execution.

A hybrid vulnerability analysis approach has been discussed in the security topic taint analysis. Taint analysis approach has been widely applied in *ex post facto* security applications, such as computer forensics, attack provenance investigation, and reverse engineering. Ming et al. [134] developed StraightTaint, a hybrid taint analysis tool that decouples the program execution and taint analysis. In order to test the accuracy of their approach/tool, they used CVE data to test the accuracy of the taint analysis task in terms of software attack detection.

Among the surveyed papers, research includes work on locating vulnerable source code, identifying attack surface, and prioritizing vulnerability impacts using static analysis tools (e.g., Nguyen et al. [133], Theisen et al. [135], Renatus et al. [136], and Sampaio et al. [52]). When a vulnerability is disclosed, it may impact organizations which rely on retro versions of the software. Nguyen et al. [133] proposed an automated method to determine the code evidence for the presence of vulnerabilities in retro software versions. Identifying the vulnerable code in retro versions is based on identifying the lines of code that were changed to fix vulnerabilities. To

[34] https://samate.nist.gov/SRD/

show the scalability of the method, the authors performed experiments on Chrome and Firefox (spanning 7, 236 vulnerable files and approximately 9, 800 vulnerabilities) on NVD. Theisen et al. [135] proposed an approach that approximates attack surfaces with stack traces. An approach for identifying vulnerable code is to identify its attack surface, the sum of all paths for untrusted data into and out of a system. The experiments were conducted on datasets collected for Windows 8. However, the dataset used in this experiment to characterize the vulnerabilities in Windows is data from NVD. Sampaio et al. [52] discussed context-sensitive data flow analysis for early vulnerability detection. The authors showed two goals of their proposed approach: (1) they proposed to perform continuous detection of security vulnerabilities while the developer is editing each program statement, also known as early detection; and (2) they explored context-sensitive data flow analysis (DFA) for improving vulnerability detection and mitigating the limitations of pattern matching. They used the OWASP repository as the source for vulnerability information for their method implementation. Renatus et al. [136] provided a tool to improve prioritization of software weaknesses using security models. The authors introduced a lightweight tool, the Augmented Vulnerability Scoring (AVUS) tool that adjusts context-free ratings of software weakness according to user defined security model. The tool is based on information from CWE databases and CVSS scoring schema.

Wang et al. [74] and Gao et al. [75] used static analysis approaches to diagnose and resolve memory vulnerabilities and to detect buffer-overflow issues, respectively. Wang et al. [74] proposed an approach for detecting memory corruption at the binary level by identifying the misuse of input data and presented an exploit diagnosis approach called deExploit. The authors evaluated their approach with several binary programs extracted from a publicly available exploits database (Exploit-DB). Gao et al. [75] presented BovInspector, a tool framework for automatic static buffer overflow warnings inspection and validated bugs repair. The tool takes two inputs—program source code and vulnerability warning—and performs a warning reachability analysis. The tool was evaluated against selected vulnerable projects from different vulnerability sources [75].

Researchers in SE use static and dynamic analysis approaches to locate vulnerable code and analyze known security vulnerabilities (e.g., XSS and CSRF). A common use of SVDBs in this context is using application vulnerabilities for validation and to gain security knowledge.

3.4.4 Security Testing

Testing can be classified utilizing the three dimensions of flow, scales, and characteristics [143] shown in Figure 10. Test flow explains where tests are derived from, scripting/coding (white-box) or requirements (black-box). Test scale describes the granularity of the system under test (SUT) and can be unit testing or anything up to system testing. The test characteristics are the reason or purpose for designing and executing a test, for example testing the system's functionality, robustness, capacity, or usability.

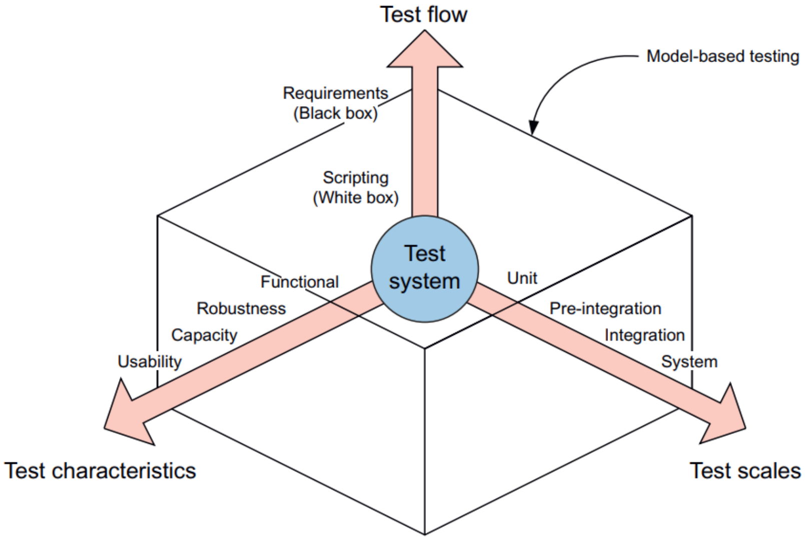


Figure 10: Types of testing in SE [143].

Model-based testing is "*the automation of test design*". Tests are generated automatically from a model of SUT [144]. Because test suites are derived from models and not from source code, model-based testing is usually seen as a form of black-box testing [144].

Identifying vulnerabilities and ensuring security functionality by security testing is a widely applied measure to evaluate and improve the security of software. Software security testing is a process intended to reveal flaws in the security mechanisms of information systems that protect data and maintain functionality as intended [145].

Our survey provides an overview of the security testing techniques used in SE research. Table 8 summarizes the articles classified in this SE task, showing the title of each article, publication year, name of the used SVDB, and what the SVDB was used for.

Table 8: SE articles using SVDBs in security testing task

Reference	**Title of Paper**	**Year**	**SVDBs**	**How SVDBs Used**
Stivalet et al. [78]	Large Scale Generation of Complex and Faulty PHP Test Cases	2016	CWE and OWASP	Used vulnerabilities information for comprehending security attacks.
Ceccato et al. [76]	SOFIA: An Automated Security Oracle for Black-Box Testing of SQL-Injection Vulnerabilities	2016	CVE	Downloaded vulnerable web applications and web services affected by SQL injection.
Pham et al. [77]	Model-Based Whitebox Fuzzing for Program Binaries	2016	OSVDB and CVE	Downloaded vulnerable projects.
Palsetia et al. [53]	Securing Native XML Database-Driven Web Applications from XQuery Injection Vulnerabilities	2016	OWASP	Used security guidelines.
Bozic et al. [54]	Evaluation of the IPO-Family Algorithms for Test Case Generation in Web Security Testing	2015	OWASP and Exploit-DB	Downloaded vulnerable web applications.
Appelt et al. [146]	Behind an Application Firewall, Are We Safe from SQL Injection Attacks?	2015	OWASP	Used the database as a knowledge source to comprehend SQL injection attack.
Pham et al. [147]	Hercules: Reproducing Crashes in Real-World Application Binaries	2015	CVE	Downloaded selected CVEs vulnerabilities reports.
Aydin et al. [148]	Automated Test Generation from Vulnerability Signatures	2014	OWASP	Downloaded vulnerable web applications.
Hossen et al. [149]	Automatic Generation of Test Drivers for Model Inference of Web Applications	2013	OWASP	Downloaded vulnerable web applications.
Blome et al. [150]	VERA: A Flexible Model-Based Vulnerability Testing Tool	2013	OWASP	Downloaded vulnerable web applications.
Lebeau et al. [151]	Model-Based Vulnerability Testing for Web Applications	2013	CAPEC and OWASP	Used vulnerability knowledge and types of security attacks.
Buchler et al. [152]	SPaCiTE -- Web Application Testing Engine	2012	OWASP	Downloaded Webgoat, a vulnerable web application.
Zhang et al. [153]	SimFuzz: Test Case Similarity Directed Deep Fuzzing	2012	NVD, SecurityFocus, and SecurityTracker	Extracted 100 buffer overflow vulnerabilities.
Shahriar et al. [154]	MUTEC: Mutation-Based Testing of Cross Site Scripting	2009	OSVDB	Downloaded five open source web applications suffering from Cross Site Scripting (XSS) vulnerability.

The surveyed papers in this SE task used SVDBs information to perform security testing and techniques in different aspects such as black-box testing (including model-based testing) as discussed in Ceccato et al. [76], Palsetia et al. [53], Appelt et al. [146], Aydin et al. [148], Hossen et al. [149], Blome et al. [150], Lebeau et al. [151], and Buchler et al. [152].

Black box and model-based testing. Black-box and model-based security testing are testing techniques that describe how a system securely behaves in response to an action (determined by a model). Ceccato et al. [76] introduced SOFIA, a security oracle for black-box testing of SQL-injection vulnerabilities. The main purpose of SOFIA is to detect types of SQLi attacks. The proposed approach validated vulnerable web applications that use SQL relational database and has known vulnerabilities published in the CVE dataset.

Palsetia et al. [53] proposed a black-box fuzzing approach to detect different types of XQuery injection vulnerabilities in web applications driven by "native XML databases"[35]. The primary objective of the proposed method for detecting XQuery injection vulnerabilities is based on OWASP guidelines in native XML database-driven web applications.

Appelt et al. [146] focused on web application firewalls and SQL injection attacks. They presented a black-box (machine learning-based) testing approach to detect holes in firewalls that let SQL injection attacks bypass. They developed a tool that implements the approach and evaluated it on ModSecurity[36], a widely used application firewall provided by the OWASP project.

Aydin et al. [148] showed that vulnerability signatures computed for deliberately insecure web applications (developed for demonstrating different types of vulnerabilities) can be used to generate test cases for other applications. Their proposed approach is a black-box specification-based testing approach. The authors used a deliberately insecure web application called Damn Vulnerable Web Application (DVWA) listed in the OWASP broken web applications project.

Hossen et al. [149] proposed automatic generation of test drivers for model inference of web applications. The authors have applied the method on WebGoat. WebGoat is organized in lessons; the goal of a lesson is to show a type of vulnerability and its corresponding attack. The

[35] http://www.rpbourret.com/xml/ProdsNative.htm

[36] https://www.modsecurity.org/

authors chose the stored cross-site scripting (XSS) lesson, which has a demonstrated vulnerability in the Top10 within OWASP.

Blome et al. [150] introduced VERA—a flexible model-based vulnerability testing tool. The proposed method is a tool that allows users to define attacker models where the payloads and the behavior are separated and that abstract away from low-level implementation details such as HTTP requests. The researchers give two examples of Injection flaws: Cross Site Scripting and SQL Injection using information from WebGoat extracted from the OWASP database. Lebeau et al. [151] introduced model-based vulnerability testing for web applications. The approach is based on a mixed modeling of the application under test; the specification indeed captures some behavioral aspects of the web application and includes vulnerability test purposes to drive the test generation algorithm. This approach is illustrated with the widely-used DVWA example.

Buchler et al. [152] presented a model checking tool called SPaCiTE - Web Application Testing Engine. The proposed tool relies on a dedicated model-checker for security analyses that generates potential attacks with regard to common vulnerabilities in web applications. The authors applied SPaCiTE to Role-Based-Access-Control (RBAC) and Cross-Site Scripting (XSS) lessons of WebGoat, an insecure web application maintained by OWASP.

Hybrid approaches. We found papers that used a combination of different testing approaches, such as model-based white box testing approach (e.g., Pham et al. [77]), and an article that combined different approaches such as black box fuzzing, code analysis, and combination (i.e., combinatorial) testing (e.g., Zhang et al. [153]).

Pham et al. [77] presented Model-based Whitebox Fuzzing (MoWF), an automated testing technique for program binaries that process structured inputs. MoWF is a combination of model-based black box fuzzing and white box fuzzing. They evaluated their approach on 13 vulnerabilities in 8 program binaries with 6 separate file formats, and compared the explored vulnerabilities with ones from OSVD and CVE public databases.

Zhang et al. [153] proposed a fuzzing approach aimed to produce test inputs to explore deep program semantics. The fuzzing process integrates techniques from black-box fuzzing, code analysis, and combination testing. The main purpose of the approach is to detect memory corruption vulnerabilities such as buffer overflow and pointer out-of-boundary operations. The authors used top 100 buffer overflow vulnerabilities from searching the vulnerabilities of NVD, SecurityFocus, and SecurityTracker.

Others. Other papers used SVDBs information in testing approaches such as the combinatorial testing approach discussed in Bozic et al. [54], and symbolic execution testing explained in Pham et al. [147]. In this category are also mutation-based testing introduced in Shahriar et al. [154] and test case generator tool for PHP introduced in Stivalet et al. [78].

Bozic et al. [54] evaluated in-parameter-order (IPO-Family) algorithms, namely the IPOG and IPOG-F algorithms, for test case generation in web security testing by using a combinatorial testing approach. They validated the proposed approach on vulnerable web applications published in the OWASP Broken Project[37] and in the Exploit Database[38].

Pham et al. [147] presented the design and evaluation of the *Hercules* approach for finding test inputs which can reproduce a given crash. The approach is based on symbolic execution and its distinctive features. The test input generated by their method serves as a witness of the crash. They illustrated the pertinent aspects of the approach using data from the CVE database.

Shahriar et al. [154] introduced an approach called MUTEC, a Mutation-based Testing of Cross Site Scripting. The approach tries to address XSS vulnerabilities related to web-applications that use PHP and JavaScript code to generate dynamic HTML contents. Shahriar et al. proposed 11 mutation operators to force the generation of an adequate test dataset. The proposed operators were validated by using five open source applications having XSS extracted from the OSVDB repository.

Stivalet et al. [78] presented an automated generator of test cases, which are designed to evaluate source code security analyzers. The tool produces PHP programs with most common vulnerabilities embedded in various code complexities. The authors used CWE weakness dataset and OWASP vulnerabilities categories to generate selected PHP programs test cases. The generated PHP test cases were added to the Software Assurance Reference Dataset (SARD).

[37] https://www.owasp.org/index.php/OWASP_Broken_Web_Applications_Project

[38] https://www.exploit-db.com/

Among the articles being classified under this SE task (i.e., software security testing), the common use of SVDBs is to generate security test cases and validate the testing approaches on real-world vulnerable applications.

3.4.5 Risk Analysis and Management

Some software development projects still have high failure rates [155]. A diversity of risk management approaches are suggested by researchers and followed by organizations in order to minimize the failure rate and ensure project success [155]. In particular, risk management is all about perception and detection of sources of risks through the different phases of software development [155]. Table 9 summarizes the articles classified in this SE task, showing the title of each article, publication year, name of the used SVDB, and what the SVDB was used for.

Table 9: SE articles using SVDBs in risk analysis and management

Reference	**Title of Paper**	**Year**	**SVDBs**	**How SVDBs Used**
Plate et al. [156]	Impact Assessment for Vulnerabilities in Open-Source Software Libraries	2015	NVD	Downloaded specific vulnerable releases.
Yu et al. [157]	Automated Analysis of Security Requirements Through Risk-Based Argumentation	2015	CWE and CAPEC	Used vulnerability mitigation guidelines.
Cox et al.[158]	Measuring Dependency Freshness in Software Systems	2015	CVE	Downloaded and extracted vulnerability information such as disclosure date.
Kannavara et al. [159]	Assessing the Threat Landscape for Software Libraries	2014	CWE and CAPEC	Downloaded types of vulnerabilities attacks and patterns.
Kannavara et al. [160]	Securing Opensource Code via Static Analysis	2012	NVD	Map defined vulnerabilities to known vulnerabilities in NVD entries.
Houmb et al. [161]	Quantifying Security Risk Level from CVSS Estimates of Frequency and Impact	2010	CVSS	The authors used severity score system.
Fruhwirth et al. [162]	Improving CVSS-Based Vulnerability Prioritization and Response with Context Information	2009	NVD	Extracted severity scores from sample number of vulnerabilities reports.
Boldt et al. [163]	Software Vulnerability Assessment Version Extraction and Verification	2007	OSVDB	Found publicly disclosed vulnerabilities for specific projects.

The surveyed articles in this SE task focused on assessing security vulnerability threats and impacts on software systems.

Open source threat and impact assessment. We found 3 articles (Plate et al. [156], Kannavara et al. [159], and Kannavara et al. [160]) discussing the risk of adopting open source components (e.g., libraries) in the development environment. For example, Plate et al. [156] proposed an approach to support the impact assessment based on the analysis of code changes introduced by security fixes of open source libraries. For the evaluation, the authors depended on vulnerabilities from the NVD database. Kannavara et al. [159] sought to assess the threat landscape associated with software open source libraries and discussed mitigation strategies via Security Development Lifecycle (SDL). The used datasets in this research were from CWE and CAPEC for common vulnerability attack patterns. Kannavara et al. [160] attempted to some extent to address open source code security challenges by applying static analysis on a popular open source project (i.e., Linux kernel). Based on their analysis, the authors proposed an alternate workflow that can be adopted while incorporating open source software in a commercial software development process. For the vulnerability analysis part, the authors used CVE information for Linux kernel in the NVD repository.

Assessing security requirement. Yu et al. [157] discussed automated analysis of security requirements through risk-based argumentation. The authors' earlier work on RISA (RIsk assessment in Security Argumentation) showed that informal risk assessment can complement the formal analysis of security requirements. They incorporated an automated search functionality to match catalogues of security vulnerabilities such as CAPEC and CWE with the keywords derived from the arguments.

Vulnerability impact and severity assessment systems. We found two articles (Houmb et al. [161] and Fruhwirth et al. [162]) discussing the CVSS system risk estimation and improvements. The CVSS aids in such prioritization by providing a metric for the severity of vulnerabilities. For example, Houmb et al. [161] presented a risk estimation approach that makes use of one such data source, the CVSS. The CVSS Risk Level Estimation estimates a security risk level from vulnerability information as a combination of frequency and impact estimates derived from the CVSS. Fruhwirth et al. [162] introduced an approach for improving CVSS-based vulnerability prioritization and response with context information. They claim the CVSS scores provided by the NVD alone are of limited use for vulnerability prioritization in practice. They presented a method that enables practitioners to estimate missing context information (improvements).

Last, we find articles assessing the software versions security and updates (e.g., Boldt et al. [163] and Cox et al. [158]). Boldt et al. [163] introduced a method for software vulnerability assessment—version extraction and verification. A tool is proposed for identifying relevant version information and for verifying potential threats matched against a software vulnerability database (based on OSVDB). Cox et al. [158] aimed to make prioritization of the software libraries updates more transparent by introducing metrics to quantify the use of recent versions of dependencies (i.e., the system's "dependency freshness"). They validated the usefulness of the metric using interviews, analyzed the variance of the metric through time, and investigated the relationship between outdated dependencies and security vulnerabilities reported in the CVE database.

The common use of SVDBs within this SE task is to investigate vulnerabilities impacts on OSS libraries, to elicit security requirements through risk-based argumentation, and vulnerability prioritization based on severity impact.

3.4.6 Other Tasks

Our survey also showed that SE researchers used SVDBs for domains not associated with any of our manual classes mentioned in the above sub-sections. Mendes et al. [56] proposed a methodology for benchmarking the security of software-based systems. The vulnerabilities information for the subject systems were crawled from OSVDB and NVD data sources. Min et al. [79] proposed a technique for counteracting web browser exploits. The approach was validated on two vulnerable applications versions (Flash and Adobe Reader) extracted from the CVE database.

3.5 Study Implications

In this section, we organize the results of our mapping study of the surveyed articles and present a discussion on the common pitfalls when using SVDBs for SE tasks.

3.5.1 How the Surveyed Articles Used SVDBs for Different SE Tasks

For each task, we look at two dimensions: (1) the SVDB that is used, and (2) the repositories that are used along with the SVDB. In other words, given a SE task, we want to answer the questions of which SVDB is usually used and which SE repositories are often used along with SVDBs. The results may help new researchers (and practitioners) determine how to best use SVDBs to a particular SE task.

Table 10 shows how the surveyed articles support each of the SE tasks. We focus on the six tasks that we previously identified (see Table 4), and we show the percentage of the surveyed articles that used each kind of SVDBs and SE repository. We found that most articles reported on the use of *common* SVDBs compared to *specialized* SVDBs. The reasons for this are manifold such as: (1) *Specialized* SVDBs contain known security vulnerabilities affecting specific systems written in a specific programming language (e.g., PHP). Analysis results obtained from *specialized* SVDBs are typically not generalized to other systems (e.g., using Java vs PHP), therefore limiting the potential impact of the published work. (2) *Common* SVDBs contain more diverse known security vulnerabilities affecting different types of software systems and therefore can accommodate different research interests. (3) Among the *common* SVDBs, we found NVD to be the most popular SVDB used in the SE community. There are several reasons for the popularity of NVD including ease of access (e.g., automatic data feeds), updates, size, and quality of the dataset.

Table 10: Summary of how surveyed articles used SVDBs for different SE tasks. The numbers are shown in percentage for each category (i.e., SVDBs types, and SE Repo. used).

		Empirical Research	Modeling	Testing	Vulnerability Analysis	Risk Analysis	Other SE Tasks
SVDBs Types	Common	47	22	15	13	9	3
	Specialized	2	0	0	2	0	0
SE Repo. Used	Source Code	15	10	8	13	3	1
	Bugs / Vuln. Reports	29	9	2	2	2	2
	Logs	2	1	2	0	0	0
	Req./Desgin	1	2	0	0	3	0
	Other SE dataset	1	0	3	0	1	0

Even with the popularity of *common* SVDBs, studies have shown that developers are often not aware of known security vulnerabilities affecting their systems [156], [164], [165], resulting in situations where known vulnerabilities are only late or never patched after the disclosure of a vulnerability. This implies limited communication between vendors in charge of patching the vulnerabilities and *common* SVDBs providers, since vendors are expected to provide a new (patched) version of components with known vulnerabilities or at least provide users with patch information on how to fix the vulnerabilities.

A limitation of many *common* SVDBs is that they do not include the actual code causing the security vulnerability, which is in contrast to *specialized* SVDBs that often share the code of known security vulnerabilities. Having direct access to this vulnerable code fragment simplifies the work of SE researchers evaluating their security analysis approaches.

We find that most tasks only analyze source code and bugs/vulnerability reports (i.e., issue tracker data) and rarely use other repositories. With the open source community and its supporting ecosystem providing access to its source code and bug repositories, the security research community takes advantage of these available knowledge resources to analyze known vulnerabilities reports and link vulnerabilities reported in SVDBs with available open source issue tracker or version control systems.

We reviewed SE tasks discussed in SE articles that used SVDBs in their research methodology to identify how these SVDBs are used. We classified the articles based on describing SE tasks for a more fine-grained analysis. Our findings reveal that empirical research such as security studies (e.g., case studies, comparison studies) are among the most common research activities covered by our reviewed articles. Although some research has shown that combining multiple SVDBs can improve vulnerability detection coverage and performance [91], [166], we found most articles cited only a single SVDB. Also we found most SVDBs host vulnerabilities affecting commercial (or closed-source) applications, but most of the use case studies were conducted on open source applications such as Apache project[39], Chromium project[40], and Mozilla open source project[41]. We believe the reason for this is that the open source project provides rich information resources (e.g., issue tracker data, source code and version history, email archive, etc.) which can be used along with SVDBs information. This

[39] https://projects.apache.org/

[40] https://www.chromium.org/

[41] https://developer.mozilla.org/en-US/docs/Mozilla/Developer_guide/Introduction

helps to study the complete development environment for analyzing security vulnerability (i.e., evolution). This advantage of gaining such application's information is not always available for commercial (or closed-source) applications.

The second most common SE task covered by our reviewed articles is modeling. The common use of SVDBs in this task (e.g., [129]) is to apply a vulnerability model to a number of well-known vulnerabilities. In general, this has resulted in comprehensive understanding of the vulnerabilities and the measures required to prevent them.

Lastly, testing task is also a common SE task covered by our survey articles. Most of the testing approaches were conducted on web-applications (e.g., [54], [76], [151], [152]) and, more specifically, validating testing approaches on injection attack (e.g., [73], [76], [146]). One of the reasons the injection attack is classified as 1st most critical web application security risk is that it has been confirmed as an OWASP Top 10 [42] attack type. However, SVDBs attracted SE researchers in this domain (testing) due to the rich information provided by SVDBs regarding this type of attack. For example, in 2018 NVD host 6.09% injection attack, ranging from SQL injection 2.56% (inject SQL commands that can read or modify data from a database) to OS command injection 0.66% (full system compromise).

Summary: *The common use of SVDBs in these SE tasks is extracting vulnerability examples for validating the assumptions proposed by authors and comprehending the security vulnerability affecting the software system. Also, studies on vulnerability repositories focus on harvesting statistical trends or creating vulnerability models and using them for prediction. Other studies focus on the vulnerability reporters who possess the most important information.*

3.5.2 Common Pitfalls when Using SVDBs in Software Engineering Tasks

In Section 3.4, we discussed how SVDBs are commonly used in different SE tasks. In this subsection, we further discuss the common pitfalls when using SVDBs in SE tasks.

[42] https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project

3.5.2.1 Vulnerability disclosure date

Although researchers in SE have proposed several approaches ranging from empirically studying the vulnerability evolution to vulnerability prediction models, we see very few studies that consider the various vulnerability sources in order to avoid bias threats to the validity of their approaches. Determining the public disclosure date of a vulnerability is vital to understanding the timeline of a vulnerability's life cycle. Previous studies (e.g., [102], [167], [168], [106], and [169]) relied on a single SVDB (e.g., NVD) as a source for their empirical studies on vulnerability life cycle and security patches. But SVDBs entries contain a CVE publication date that corresponds to when the vulnerability was published in the database, not necessarily when it was actually publicly disclosed. For example in our surveyed articles, Munaiah et al. [82] and Murtaza et al. [86] relied on a single SVDB source for their vulnerability empirical analyses. Munaiah et al. [82] used NVD and relied on the vulnerability disclosure date provided by NVD only. In the same way, Murtaza et al. [86] used NVD vulnerabilities release dates without considering other SVDBs. However, this may affect the proposed approaches on vulnerability lifetime analysis, which would in turn affect the authors' conclusions.

Current studies usually use SVDBs as a black box and do not consider the effect of using different SVDBs information on the SE task. As a result, future studies may want to examine the effect of different SVDBs information on SE tasks. For example, there is no clear guideline on how estimating the vulnerability disclosure date may affect the result of vulnerability prediction models. Providing such a guideline can help SE researchers choose better vulnerability disclosure dates.

3.5.2.2 Vulnerability information noise

When using SVDBs for tasks such as tracing security vulnerabilities, unstructured vulnerability information in some SVDBs may affect the vulnerability extraction and linking process. For example, unstructured vulnerability information in SVDBs requires text mining techniques and human labor in order to detect patch information and locate vulnerability causes in the source code. Since this relies on the vulnerability auditors and reporters themselves, we do not currently see a way to enforce this. Other vulnerability representation-related solutions include those that synthesize exploit code examples for specific vulnerability or those that try to present existing vulnerability online resources in ways more useful to developers (e.g., [165]).

3.5.2.3 Lack of knowledge

Sometimes researchers may not be aware of different existing public SVDBs. Due to that lack of knowledge, researchers might miss important vulnerability information which is already known. For example, Scandariato et al. [61] showed detailed background about SVDBs (i.e., NVD); however, they used the Fortify SCA tool to identify vulnerabilities via static source code analysis rather than using the vulnerabilities reported in a database such as the NVD. For that, the authors claimed that "*the choice was obligatory, as there are no public databases with sufficient numbers of vulnerabilities to analyze for Android application*". Although Android vulnerabilities were rarely published in NVD at the time, they were published and discussed in other SVDBs very often; for example, Exploit-DB[43] and vulnerability-lab[44] host over 200 android apps vulnerabilities.

However, relying on vulnerability scanner tools to extract vulnerabilities without (or instead of) using known vulnerabilities published in different SVDBs may increase the threats to validity of a researcher's approach and increase the effort in evaluation.

3.6 Threats to Validity

In what follows we discuss external and internal threats to validity for our study and how we attempted to mitigate them.

External Threats: Literature search and selection process. Our online library search is based on keyword queries that include terms related to vulnerabilities databases and SE. It is possible that our search omitted some studies that either implicitly used vulnerabilities databases without mentioning the term 'vulnerability', or papers that described the use of vulnerabilities databases in SE activities which were not covered by our search terms. To mitigate this threat, in addition to the online search we performed a manual issue-by-issue search covering articles published in

[43] https://www.exploit-db.com/
[44] https://www.vulnerability-lab.com/

major SE conferences and journals between 2001 and 2017. The manual search allowed us to identify those implicit ‘vulnerabilities database’ papers and those papers that leverage vulnerabilities database for SE activities not covered by our Web-based keyword search. We also used citations in relevant papers to further extend our dataset to include other relevant articles.

Internal Threats: Literature classification. We manually classified all 94 papers into 6 categories based on their study types and their targeted SE domains/tasks. There is no ground truth labelling for such classification. Even though we referred to the ACM Computing Classification System, the IEEE Taxonomy of Software Engineering Standards, and the 2014 IEEE Keywords Taxonomy, there is no well-defined standard methodology of classification regarding the used schemes. To minimise any potential classification error, we carefully analysed the full text of the collected papers and had three Ph.D. students from our lab perform a cross validation of the classification, reaching an average inter-rater agreement of 91%. The disagreed/controversial papers were resolved through further discussion.

3.7 Future Research Opportunities

In this section, we discuss some opportunities for future work on using SVDBs to support SE tasks.

Using SVDBs beyond just being information silos. Our results show that a majority of SE researchers use SVDBs to gain security related knowledge. In fact, developers already use SVDBs to identify security vulnerabilities and determine features (e.g., vulnerability patch information) that they want to implement. Presently, the role of SVDBs is mostly as a repository for reporting known security vulnerabilities. However, we envision that future versions of SVDBs will play an increasing role as an integrated knowledge source for guiding secure software development, providing security testing, and refining software security design. Hence, we believe that future versions of SVDBs need to incorporate a mechanism through which SE researchers can link and trace vulnerability information directly across knowledge resources.

Another interesting finding is that SE researchers reuse vulnerability information usually only from one source (single SVDB), thus limiting their analysis approach to the data available in this SVDB. One approach to address this problem is to improve the accessibility of information across SVDBs boundaries. Providing users with standardized access to these knowledge resources, where queries will retrieve information across SVDBs boundaries, will represent a first step to performing new types of vulnerability analyses (e.g., global security impact). While linking these knowledge resources is an important initial step, additional semantic modelling will be needed to ensure the consistency and quality of knowledge across SVDBs boundaries. For example, threats to consistency and ambiguity across these knowledge resources will have to be addressed to ensure that a vulnerability reported in two databases is actually the same (or different) instance. One approach would be to replace current proprietary knowledge modelling approaches used by SVDB providers and agree upon a standardized knowledge modelling approach, which would include the ability to semantically link query SVDBs across the repository boundaries and to provide each vulnerability with a global, unique identifier, similar to the Universal Resource Identifier used by the SW.

Linking Security Commit Changes to SVDBs. With more widespread use of SVDBs in software development, we believe that SVDBs should become an integrated part of current software development processes and best practices. Similar to the current practice of adding an issue number to a commit message, a commit message also should include a link to the vulnerability in the SVDB where it is reported. Such vulnerability traceability can provide additional insights and documentation to QA[45] and future maintainers when analyzing and comprehending the code patch. Furthermore, a bi-directional link from the vulnerabilities to the known and patched code would be desirable. We further believe that next generation IDEs should not only facilitate this linking process, but also take advantage of these links to recommend patches or identify potential impacts of these vulnerabilities on other parts of the system.

Vulnerability scanner tools using SVDBs. The existing vulnerability detection tools provide too much detail about the vulnerability issue and these details, sometimes, are unnecessary to the regular developer which affects the vulnerability comprehension. We do not see this as something that can be enforced, but at least guidelines can be developed to help regular developers achieve vulnerability patches easily.

[45] Quality Assurance

The unified model for the public SVDBs (suggested above) can be used to enhance the vulnerability scanner tools to reduce false positive and negative results. Thereby, we suggest the following list of tool features that encourage future tools or solutions design to be considered:

- Support at least the following tasks: tracing the same vulnerability from different SVDBs, summarized report about the vulnerability and its impacts, provide patch information if it exists, automatically deploy the patch to the vulnerable project if needed, vulnerability dependencies and its global impact (find relevant libraries that might be affected with the same vulnerability).
- Given a piece of code that introduces the vulnerability, identify any internal or external calls for this vulnerable code.

3.8 Chapter Summary

While the SE research community is increasingly focusing on security and reliability, no comprehensive literature survey exists that studies how SVDBs are used and integrated in the SE life cycle. Knowing how researchers use SVDBs may also help future studies improve software security. In this study, we surveyed 94 articles from the SE literature that used SVDBs. We found that:

- there is an increasing awareness of SVDBs in the research community in terms of papers being published describing the use and application of SVDBs in the SE domain;
- the majority of the surveyed studies applied SVDBs only to a limited number of SE activities;
- most studies relied on only one SVDB for their contribution.

We have also discussed potential directions for future work on using SVDBs for different SE tasks, which also were the motivation for our research presented in Chapters 4–7 of this thesis.

In the next chapter, we discuss in more detail the knowledge engineering process we applied to create a unified ontological representation for SVDBs, which forms the basis for a more seamless integration of SVDBs into existing SE development tasks.

Chapter 4

4 A Unified Ontology-based Modeling Approach for Software Vulnerabilities Data Sources

The ultimate goal of constructing a vulnerability's knowledge base is to enable stakeholders (e.g., developers) to realize their tasks, through automatically linking SVDBs to allow for an effective use of software vulnerability information. Developing such a vulnerabilities' knowledge base is a complex task, since the ontologies used for the data representation have to be sufficiently expressive and flexible to allow for knowledge reuse and sharing, as well as support for different SE tasks. In this chapter, we explain our knowledge engineering methodology which we applied to the construction of our unified vulnerability information knowledge base and the design decisions we made to address some of the open research challenges identified in our literature review (Chapter 3).

4.1 Introduction

Vulnerabilities in software systems are one of the primary causes for security threats and breaches. These vulnerabilities not only affect the usability of these affected systems, but software productivity and competitiveness are also increasingly dependent on the successful management of vulnerability information. In order to address the removal and management of security threats, the security community has introduced several vulnerability knowledge sources, such as Security Vulnerabilities Databases (SVDBs) that capture information about software vulnerabilities. However, with the amount of security vulnerability data available and with this information being spread across heterogeneous SVDBs, software developers are struggling to take full advantage of these SVDBs. The situation is further complicated by the fact that these SVDBs also introduce ambiguity into their datasets, not only in terms of *what* but also *how* this vulnerability information is modeled in the databases. The heterogeneity and ambiguity of SVDBs leads to diverse data modeling results and has become a major challenge for organizations managing both disclosure and access to vulnerability information. In addition, while individual SVDBs provide access to their information through APIs, RSS feeds, or notification services, the sharing and integration of the information across these resources remains an open challenge.

The Semantic Web (SW) and its enabling technology stack have been introduced to address knowledge sharing across heterogeneous data resources using ontologies. Such a unified ontological representation forms the basis for queries and data analysis across knowledge boundaries, removing some of the traditional information silos that existing vulnerability knowledge resources have maintained. However, the security domain currently lacks ontologies that can be readily reused, shared, and deployed to conceptualize and standardize the domain of software vulnerabilities. The few existing ontologies (e.g., [170]–[172]) are limited in terms of their modeling, by focusing mostly on modeling system level (proprietary) vulnerability information related to a particular information resource rather than trying to capture software vulnerability information as a domain of discourse.

In this chapter we introduce a semi-automated approach for the development of a software security vulnerabilities domain ontology, which is based on the discovery, reuse, and integration of knowledge from existing vulnerability SVDBs. More specifically, our methodology takes advantage of Formal Concept Analysis (FCA) [34] to guide knowledge engineers during the modeling process. We also deal with the ambiguity and inconsistencies found in existing vulnerability information by introducing a set of primitive operations to support the ontology alignment. In our modeling approach, we furthermore take advantage of SW inference services to infer implicit vulnerability information. We illustrate the applicability of our modeling approach by instantiating a vulnerability ontology for unifying vulnerability information from several open source SVDBs.

The main contributions of our work are as follows:

- A literature review of existing security vulnerability ontologies.

- Existing SVDBs can be considered as information silos which only allow limited data and knowledge sharing across SVDB boundaries. We propose a semi-automated methodology using FCA to create a unified ontological knowledge model that supports knowledge sharing, linking, and inference across SVDB boundaries.
- We present alignment rules to facilitate knowledge integration and improve our overall knowledge design.
- We illustrate the applicability of our modeling approach by providing examples of how our modeling approach supports vulnerability analysis across individual SVDBs.

4.2 Literature Review

In what follows, we present a detailed literature review on the use of ontologies to model software vulnerabilities. Our literature review focused on two core aspects: (1) research articles that propose, create, or use *software* security ontologies; and (2) articles that propose ontology development and integration using formal methods (e.g., formal concept analysis).

4.2.1 Software Security Ontologies

In this section, we review research involving *software* security ontologies by classifying software security ontologies based on their usage context, and the tools and methods used to develop these ontologies (i.e., the knowledge engineering they followed). The review is based on review guidelines suggested by Petersen et al. [50].

While existing literature surveys have reviewed software security ontologies for particular applications (e.g., Souag et al. [173], Blanco et al. [174], and Sicilia et al. [175]), our survey focuses on the methodologies used by existing work for creating these ontologies and how these ontologies are then applied in the software engineering domain.

Data collection. For the data collection process, we defined a set of inclusion criteria which papers had to meet to be included in our survey. The main selection criterion was that a paper must discuss the use of software security vulnerabilities ontologies. Among the other criteria which we applied during the selection process were that an article be written in English and published as a conference paper, journal paper, technical report, or book in a reputable venue. For our survey we only considered articles published between January 2005 and December 2017, more than a full decade of research results.

For our survey we conducted two online searches: (1) we used existing library and scholarly search engines (e.g., ACM Digital Library, IEEE Xplore Digital Library, Google Scholar, etc.), and (2) we performed an additional manual inspection of selected venues and papers published at these venues to identify additional papers not covered by the search engines.

For the two online searches we used the following search terms: "vulnerability", "security", "software", and "ontology". The results of the online search left us with a total of 70 unique articles. As part of our data cleaning, we manually reviewed the title and abstract (and in some cases the introduction) of the paper to verify that an article met our main inclusion criteria—*a paper must discuss the use of software security ontology*. Papers not meeting our inclusion criteria where omitted from further processing. After completing this manual review process, 44 of the initial 70 articles were considered for a more detailed review.

During the detailed review we manually verified that each paper actually discussed the use of security ontologies. Papers which did not explicitly describe the use of security ontologies for SE were removed from the set, leaving us with a total of 22 articles to be included in our final dataset. For these 22 publications we extracted and published online [176] the meta-data for each article. The extracted meta-data includes: the author(s) names, article title, publication year, publication type (academic, industry, or both), the ontology artifact created (OWL files or URL links if exist), the vulnerability data source used (i.e., SVDBs), information on the venue the paper was published, and keywords used by the article.

Results: Most papers describe different aspects of modeling, integrating, and applying software security ontologies. We classified the papers into the following categories:

(1) **Software security ontologies applied in software development**: This category uses security ontologies to capture and integrate empirical knowledge about vulnerabilities in the system development process.

(2) **Ontologies for software cybersecurity**: Cybersecurity domain ontologies are used to support information integration and cyber situational awareness in cybersecurity systems.

(3) **Ontologies for information security management**: Discuss the use of ontologies in the iterative process of identifying, classifying, remediating, and mitigating vulnerabilities.

While other classification criteria exist (e.g., software versus hardware [46] security vulnerabilities), we consider such classification categories outside the scope of our work since we are only focused on *software* vulnerabilities.

4.2.1.1 Category 1: Software security ontologies applied in software development

A significant body of work exists that discusses how ontologies in general can be integrated into software development processes. In [177], a systematic review of the application of ontologies in SE has been introduced. In [178], the authors reported on the use of ontologies in SE for different phases of the software development process (e.g., the use of ontologies to reduce ambiguity, and inconsistency in requirements). In [24], the use of ontologies to support software evolution is presented. The authors created ontologies for different software artifacts.

However, only very few papers discussed how security ontologies (knowledge) can be integrated in software engineering processes.

Kang et al. [179] presented a security ontology for identifying security requirements using an approach that combines MDA (Model-Driven Architecture) and ontologies. In their work, the authors introduced ontologies to provide security analysis at the PIM (Platform Independent Model), PSM (Platform Specific Model), and source code level.

Elahi et al. [180] proposed a vulnerability modeling ontology to integrate empirical knowledge about vulnerabilities into the system development process. The paper identifies basic concepts for modeling and analyzing vulnerabilities and their effects on the system, and uses these criteria to compare and evaluate security frameworks (such as CORAS [181] and Tropos [182]).

Souag et al. [183] introduced an ontology to support security requirements elicitation based on an earlier conducted survey [173]. The authors evaluated their approach using 10 security experts to determine whether their ontology is sufficient to support security requirements

[46] Currently, hardware vulnerabilities modeling are not in the scope of this work.

elicitation. Their study showed that providing just a security ontology might not be sufficient and that additional traceability links between requirements and a security ontology are needed.

Khoury et al. [184] presented an approach to detect security patterns using ontologies. The authors proposed a security pattern approach based on ontological mappings between requirement and design, as well as at the implementation level between threat models and bugs in the source code.

4.2.1.2 Category 2: Ontologies for software cybersecurity

In what follows, we discuss the use of ontologies as a modeling language for the cybersecurity domain. Cybersecurity is concerned with technologies, processes, and practices designed to protect networks, computers, programs, and data from attacks, damages, or unauthorized access [185].

Undercoffer et al. [186] and [187] introduced an ontology for intrusion detection systems. The proposed ontology was created based on the evaluation of 4000 vulnerabilities and the attack strategies used to exploit them. The ontology was specified using the DARPA Agent Markup Language (DAML[47]) and prototyped using DAMLJessKB [188]. The authors included several use case scenarios based on common attacks such as: Denial of Service – Syn Flood[48], the Classic Mitnick Type Attack[49], and Buffer Overflow Attack[50].

More et al. [189] presented a situation-aware intrusion detection model that integrates systems security data sources (e.g., networks logs) to create a semantically rich knowledge-base for the detection of cyber threats/vulnerabilities. The authors collected data streams from network monitors, host monitors, sensor data, and other Intrusion Detection Systems (IDS) modules, which are asserted as facts in their knowledge base (introduced by [186] and [187]). For the intrusion detection, the authors take advantage of SW reasoning services to infer whether there is an indication of an attack.

Joshi et al. [171] extracted cybersecurity-related entities, concepts, and relations, and captured them in their IDS ontology (introduced by [186] and [187], and further enhanced by [189]). As part of their approach, the authors also mapped these concepts to objects in the

[47] http://www.daml.org/
[48] https://en.wikipedia.org/wiki/SYN_flood
[49] http://wiki.cas.mcmaster.ca/index.php/The_Mitnick_attack
[50] https://en.wikipedia.org/wiki/Buffer_overflow

DBpedia[51] knowledge base using DBpedia Spotlight [190]. The approach creates a RDF linked data representation of cybersecurity concepts and vulnerability descriptions. The security information is extracted from both structured vulnerability databases and unstructured text. Their approach supports vulnerability identification and vulnerability mitigation efforts.

Syed et al. [191] introduced the Unified Cybersecurity Ontology (UCO) to support information integration and cyber situational awareness. The ontology integrates data and knowledge schema from both cybersecurity systems and commonly-used cybersecurity standards to allow for data exchange among these resources. The UCO ontology has also been mapped to a number of other existing cybersecurity ontologies ([186] and [187]) and resources on the Linked Open Data cloud ([189] and [171]).

Iannacone et al. [192] extended the existing cybersecurity ontologies from [186], [187], and [191] to provide a schema to incorporate additional information from a variety of structured and unstructured data sources.

Kamongi et al. [193] introduced VULCAN, a vulnerability assessment framework for cloud computing systems. The framework consists of two main components: an Ontological Vulnerability Assessment introduced by [194] and an Ontology Vulnerability Database [195] component. These two components provide access to known vulnerability information published by NVD. For the vulnerability assessment, the approach takes advantage of advanced reasoning capabilities to support a semantic search for vulnerabilities.

Gyrard et al. [196] and [197] introduced an ontology-based Security Toolbox for Attack and Countermeasure (STAC) to guide software developers in selecting the appropriate security mechanisms to secure Internet of Things (IoT) applications (more specifically, securing ETSI[52] Machine to Machine [M2M] architecture).

4.2.1.3 Category 3: Ontologies for information security management

Information security management approaches often rely on security ontologies for risk management, which can include automated security controls based on the Security Control Automation Protocol (SCAP[53]) to verify security compliance and security configurations.

[51] http://wiki.dbpedia.org/

[52] http://www.etsi.org/images/files/ETSITechnologyLeaflets/MachinetoMachineCommunications.pdf

[53] https://csrc.nist.gov/projects/security-content-automation-protocol

Wang et al. [172] and [198] proposed a vulnerability impact analysis using an ontology for vulnerability management (OVM). The ontology establishes the relationships between IT products, vulnerabilities, attackers, security metrics, countermeasures, and other relevant concepts. The authors illustrated the advantages of their ontologies in terms of being able to model, manage, and reason over the vulnerability information.

Wang et al. [199] and [200] proposed an approach to measure and assess how secure software products are by analyzing if they meet certain security requirements. For the assessment, the authors reused the vulnerability management ontology introduced in ([172] and [198]) to calculate an overall *environmental score*[54] for software products.

Wang et al. [201] introduced a ranking approach for attack patterns, which analyzes CAPEC (Common Attack Pattern Enumeration and Classification) associated with 14 types of CWEs. The vulnerability information is extracted from the OVM knowledge base ([172]**,** [198]) which is populated with vulnerabilities published by NVD.

Wang et al. [202] proposed a vulnerability similarity measurement that compares different vulnerabilities based on the similarity of structural hierarchies and dependencies. The information is extracted from the NVD and OVM [198] knowledge base. The similarity measure can be applied as part of different vulnerability management applications such as vulnerability classification, mitigation, and patching.

Kotenko et al. [203] and Fedorchenko et al. [204] introduced a hybrid modeling approach which extracts relationships between parts of vulnerabilities databases to create a domain ontology.

Montesino et al. [205] provided an analysis of the automation possibilities in information security management. The analysis takes into account the potential of using (i) security ontologies in risk management, (ii) hard- and software systems for the automatic operation of certain security controls, and (iii) SCAP for automating to check for compliance and security configurations.

4.2.1.4 Discussion

Literature surveys are an important aspect to understanding state of the art research in a given area, while at the same time providing directions for further studies. Our survey shows that many

[54] Environmental score represents the characteristics of a vulnerability that are relevant and unique to a particular user's environment.

such potential avenues exist for advancing the software vulnerability domain in ontology engineering. In what follows, we provide a comparison of the ontologies in the reviewed articles, using the following three comparison criteria: (1) availability of ontologies, (2) the use of a systematic knowledge engineering process for developing the ontology, and (3) the use of existing vulnerability knowledge sources (i.e., SVDBs) during the ontology development process.

Table 11: Summary of recent works on software security domain ontologies engineering

Category	**Reference**	**Available**	**Knowledge engineering process**	**Use of SVDBs**
Software security ontologies applied in software development	[179]	no	no	no
	[180]	no	no	no
	[184]	yes	no	no
	[183]	no	no	no
Ontologies for software cybersecurity	[171], [189], [186], [187], [191]	yes	no	yes
	[192]	yes	no	yes
	[193]	yes	no	yes
	[196], [197]	yes	no	yes
Ontologies for information security management	[202], [201], [199], [198], [172], [200]	no	no	yes
	[203], [204]	no	no	yes
	[205]	yes	no	yes

Our comparison (Table 11) shows that only 50% of the reviewed ontologies are publicly available online via URL links (e.g., IDS[55] owl by Joshi et al. [171]) or as a repository (e.g., cybersecurity[56] ontology by Syed et al. [191]). Based on our review, none of the surveyed ontologies specified that they used a systematic knowledge engineering approach while developing their ontologies. Most of the presented ontologies are based on the authors experience in the vulnerability domain. Our survey also shows that most papers refer to public advisories (e.g., SVDBs) as their main source of vulnerability information. However, none of the papers explain in detail how the SVDB(s) is used in their knowledge engineering methodology. Our survey also shows that SVDBs in general are not integrated in software development.

[55] http://ebiquity.umbc.edu/ontologies/cybersecurity/ids/v2.3/IDSOntology.owl
[56] https://github.com/Ebiquity/Unified-Cybersecurity-Ontology

Articles only refer to SVDBs indirectly, while describing their background section. On the other hand, SVDBs are actively used for conceptualizing software vulnerability ontologies and ontologies involved in vulnerability management. However, they are only concerned with the integration of data from different SVDBs and not on the reuse and inference of new knowledge, therefore limiting their potential applicability.

4.2.2 Ontology Development Using FCA

While work exists on describing knowledge engineering approaches using FCA for creating ontologies, these approaches have been applied to domains other than the software vulnerability domain. For example, context-based ontology building support in clinical domains using FCA [206], FCA-based ontology development for environment data integration (e.g., utility infrastructure) [207], products management and purchases control [208], information system management [209], [210], etc.

In addition, many of these formal knowledge engineering approaches either focus on knowledge reuse, knowledge integration, or just conceptualization of a domain discourse. For example, the surveyed FCA knowledge modeling approaches (e.g., [208] and [207]) are used to extract sharable knowledge in the form of upper ontologies, while our approach focuses also on the integration of software vulnerability knowledge across knowledge resource boundaries and at different abstraction levels.

4.3 Development of an Initial Software sEcurity Vulnerability ONTology (SEVONT)

Different knowledge engineering methodologies have been discussed in the literature (e.g., Noy et al. [211] and Uschold et al. [21]). For our knowledge modeling approach, we adopt a similar modeling methodology as the one presented by Noy et al. [211]. In their knowledge-engineering approach for ontology development they propose the following seven core steps: (1) determining the domain and scope of the ontology, (2) considering the reuse of existing ontologies, (3) enumerating essential terms in the ontology, (4) defining the classes and class hierarchy, (5) defining the properties of class-slots, (6) defining the facets of the slots, and (7) creating instances.

Similarly, we first conducted a thorough review of existing work on ontologies in the software vulnerability domain to help us define and determine the domain and scope of our knowledge model. As part of this review we identified and extracted key concepts from existing software vulnerability ontologies discussed in the 22 papers of our dataset using the same classification as in Section 4.2.1. Table 12 summarizes the core concepts used by the authors to construct their software security domain ontologies.

Our analysis shows that articles in the software development process category include mostly concepts related to software security requirements and requirements elicitation (e.g., dependability, confidentiality). There are cybersecurity ontologies mostly focused on Internet attacks, which typically involve different software assets that can be exploited by vulnerabilities. Ontologies used mainly for vulnerability management focus on modeling vulnerabilities knowledge sources (e.g., vulnerability databases, malware activities datasets, intrusion detections tools results, etc.) to enrich and link these knowledge resources [175].

For establishing our initial domain ontology for software vulnerabilities, we further manually identified shared (common) concepts used by these reviewed ontologies (Table 12). For example, a more detailed analysis of the ontologies and their supporting documentation shows that the *security* and *vulnerability* concepts share a similar meaning across the surveyed ontologies. These concepts are used to describe security vulnerability issues affecting software products. Also, *threat*, *attack*, *weakness,* and *risk* are concepts commonly found in these ontologies to classify software security vulnerability attacks. Based on these common concepts, we derived our initial core SEVONT ontology which includes the following concepts (Figure 11):

Table 12: Core concepts (classes) defined in the surveyed vulnerabilities ontologies

Category	**Reference**	**Core concepts**
Software security ontologies applied in software development	[179]	Security concerns, attack, frauds, asset, prevention, threats, auditing
	[180]	Vulnerability, effect, security impact, malicious action, attacker, attack, malicious goal
	[184]	Security requirements terms (confidentiality, integrity, dependability, availability, authenticity, non-repudiation, ...), security exploits, bugs, attack models
	[183]	Asset, location, organization, person, threat, vulnerability, risk severity, impact, threat agent, attack tool, attack method, security goal, security criterion, security requirement, control
Ontologies for software cybersecurity	[171], [189], [186], [187], [191]	Vulnerability, vulnerability source, product, software, hardware, operating system, web browser, consequences, means, weakness, other terms
	[192]	Software, vulnerability, malware, attack, flow, attacker, user, account, host, address, IP, port, domain name, service, address range
	[193]	Vulnerability, CVSS metric, consequences, countermeasure, IT product category, IT product, IT vendor, software, hardware, privileged program, unprivileged program, cloud type, attacker, attack intent, attack mechanism, attack
	[196], [197]	Security mechanism, security tool, security protocol, cryptographic concept, technology, attack
Ontologies for information security management	[202], [201], [199], [198], [172], [200]	Active location, introduction phase, vulnerability, IT product, IT vendor, attacker, attack, countermeasure, consequence, attack intent, attack mechanism
	[203], [204]	Weakness, attack, vulnerability, platform, countermeasure, configuration, exploit
	[205]	Control type, organization, asset, security attribute, standard control, control, threat, threat source, severity scale, vulnerability, threat origins

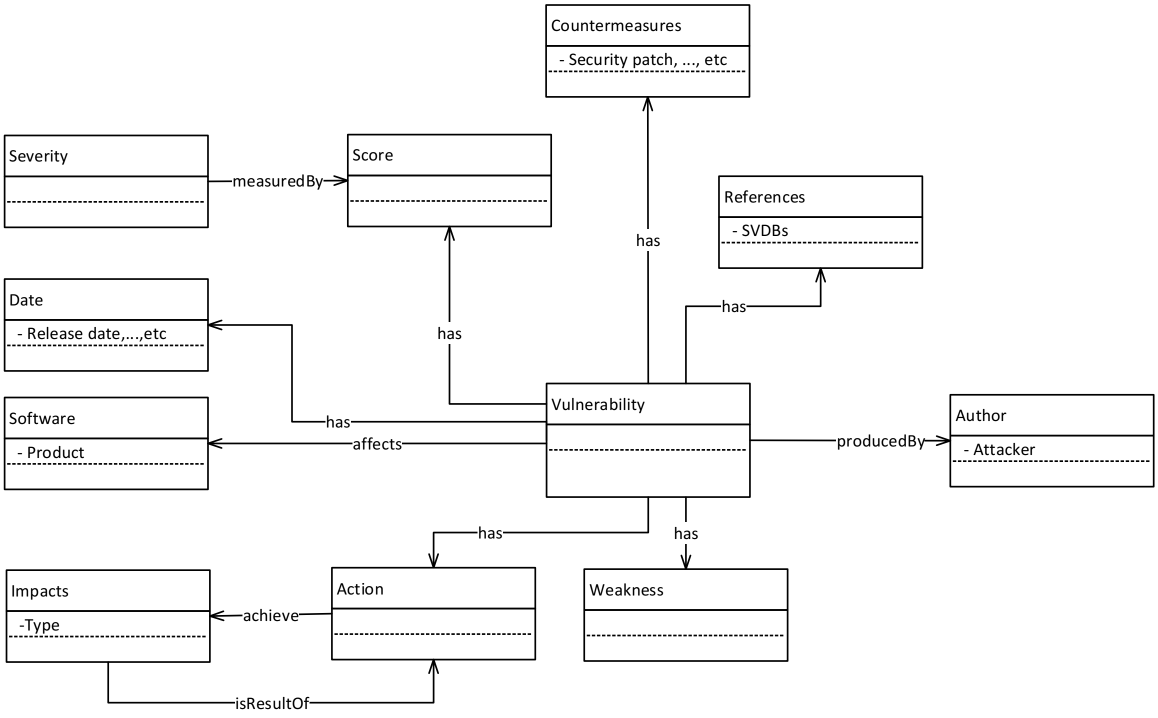


Figure 11: Initial software security vulnerability domain ontology (high-level overview).

Vulnerability. In software security, a vulnerability refers to "*an instance of a flaw, caused by a mistake in the design, development, or configuration of software such that it can be exploited to violate some explicit or implicit security policy*" [212]. Software vulnerabilities are introduced in a system by adopting a vulnerable software product, inadvertent coding mistakes by developers (e.g., bad coding practices), executing vulnerable external services (e.g., libraries), etc.

Product. This concept is used for eliciting software vulnerability information. Software product is an *Asset*—"*anything that has value to the organization*" [213], including stakeholder, information, hardware, and artifacts.

Attackers (or *malicious actor)* can be considered either internal or external entities of the system who attack a product. They perform malicious *Actions* which attempt to break the security of a software system or its components. Security databases are capturing a *Vulnerability* that has an associated *Action* and *Impact,* with an attacker exploiting a vulnerability to produce an *Action* which has an *Impact* on the system.

An *Attack* is a set of intentional unwarranted (malicious) actions designed to compromise and violate software security policies [214]. By analyzing such an attack or vulnerability pattern, analysts can study the behavior of attackers, estimate the cost of attacks, and determine their impact on overall system security.

Security analysts often rely on *References* found in SVDBs to identify vulnerable system components and to evaluate the potential *Impact* of such vulnerabilities on other parts of an ecosystem. This information is used to decide on cost-effective *Countermeasures* to eliminate a risk. When the risk of an attack is higher than the risk tolerance of stakeholders, analysts need to take an adequate *Countermeasure* to mitigate such risks.

A *countermeasure* therefore is a protection mechanism employed to secure a software system [197] (e.g., patch development, encryption/decryption enhancement, and updated system security configurations). Available information about the *Author* of a vulnerability can be used by security analysts to develop countermeasures to protect the system.

Weakness has been proposed as a concept to evaluate the impact of such an attack on the software system. *Score* is used to capture the probability of a successful attack and its *severity* on the system.

Finally, incorporating the concept of *Date* as part of a vulnerability analysis knowledge base allows for modeling the sequence of actions and exploits employed by attackers. Security analysts can take advantage of this information when designing and evaluating adequate countermeasures.

4.4 SEVONT: Knowledge Modeling and Engineering

Next, we will illustrate how our knowledge engineering approach extends on our initial SEVONT domain ontology by focusing on the reuse of existing ontologies, enumerating essential terms in the ontology, and defining classes and class hierarchies of software vulnerability concepts, relations, and properties. The outcome of this knowledge modeling process is a comprehensive ontology, capturing the domain of software vulnerability knowledge to facilitate not only the integration of heterogeneous resources but also allows for its seamless integration with other software artifacts.

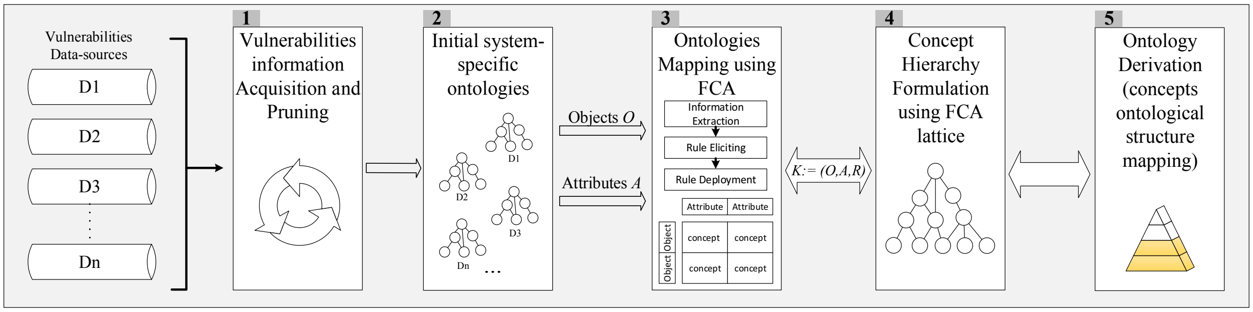

Vulnerabilities
Data-sources
D1
D2
D3
Dn
1
Vulnerabilities information Acquisition and Pruning
2
Initial system-specific ontologies
D1
D2
D3
Dn
...
Objects O
Attributes A
3
Ontologies Mapping using FCA
Information Extraction
Rule Eliciting
Rule Deployment
Attribute
Attribute
Object
Object
concept
concept
concept
concept
$K := (O,A,R)$
4
Concept Hierarchy Formulation using FCA lattice
5
Ontology Derivation (concepts ontological structure mapping)

Figure 12: An overview of our knowledge modeling methodology.

Overview: Our methodology consists of five major steps (Figure 12), based on the modeling steps presented by Noy et al. [211]. First, we extract vulnerabilities reports and meta-data published by SVDBs based on an Internet search we conduct. We then refine the scope of our model to include only "*software*" security vulnerabilities, therefore excluding hardware and configuration vulnerabilities from our model. As outlined, our knowledge modeling follows a bottom-up approach for which we take advantage of FCA to identify and abstract shared concepts from the different SVDBs. Next, we extract meta-data (attributes) from each surveyed SVDB (e.g., IDs, types, patch, timeline, etc.). For the extraction, we manually inspect concepts and properties for each SVDB and create their initial system-specific ontologies. After creating these initial system-specific ontologies, we use a combination of FCA and ontology mapping to identify and extract shared vulnerability concepts and attributes from these system-specific ontologies by creating a context table. Given this formal context table, we can now generate a concept lattice graph to visualize formal concepts which can be used by a knowledge engineer to identify shared and reusable concepts among system level ontologies. In addition, we apply a stability measure to be used by a knowledge engineer (domain expert) when deciding if a concept should remain at the system level or should be promoted to the upper levels (e.g., domain level). Finally, during the last step of our methodology, we populate our newly created knowledge base.

The model itself is based on a meta-meta model approach (e.g., Object Management Group (OMG)[57]), where the top layer captures the core elements, which are extended and refined throughout the abstraction hierarchy. Figure 13 presents an overview of the different ontology abstraction layers in SEVONT. For a complete description of these ontologies, we refer the reader to [215].

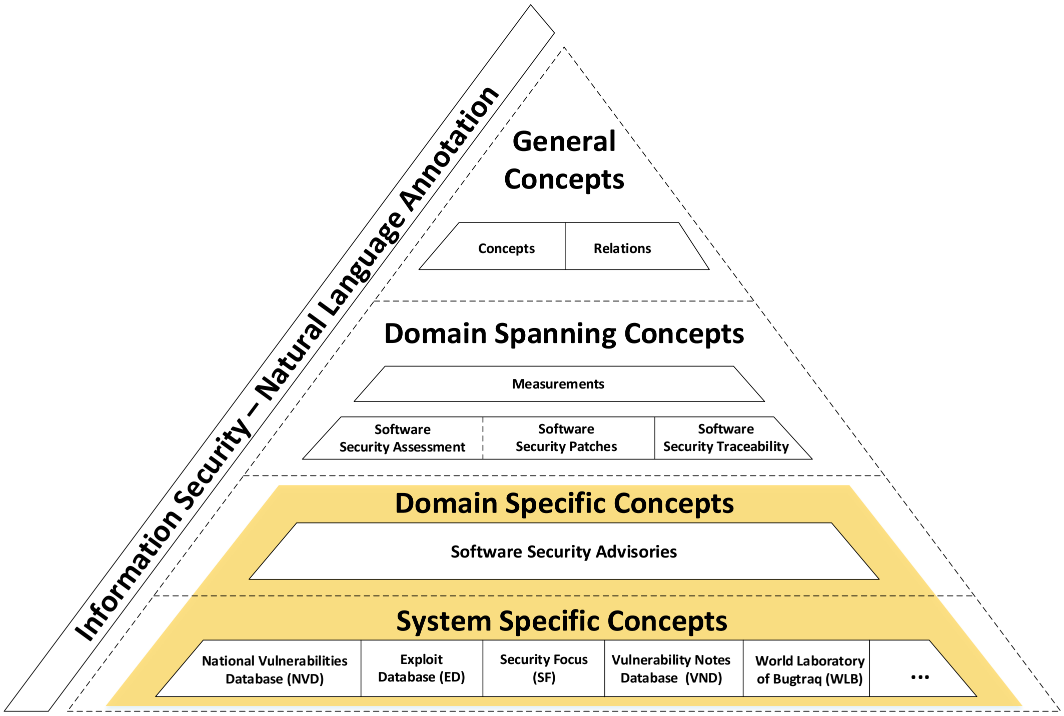

Information Security – Natural Language Annotation
General Concepts
Concepts
Relations
Domain Spanning Concepts
Measurements
Software Security Assessment
Software Security Patches
Software Security Traceability
Domain Specific Concepts
Software Security Advisories
System Specific Concepts
National Vulnerabilities Database (NVD)
Exploit Database (ED)
Security Focus (SF)
Vulnerability Notes Database (VND)
World Laboratory of Bugtraq (WLB)
...

Figure 13: The software security vulnerability analysis ontology.

Within our knowledge hierarchy, the *General Concepts* layer captures the omnipresent core concepts related to software evolution and software vulnerabilities. The *Domain-Spanning Concepts* layer builds upon the *General Concepts* layer and captures concepts that span across a number of subdomains in our model (e.g., security databases, version control systems, and source code). The concepts at the *Domain-Specific* layer are common to resources in a domain, such as software security advisory concepts. Finally, the *System-Specific* layer's concepts represent knowledge that is specific to a given data source or system and not commonly shared

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

across the domain. In Chapter 5, we discuss in detail how SEVONT can be integrated with other SE knowledge sources such as version control systems, build systems, and source code ontologies with vulnerabilities ontologies.

In what follows, we describe in detail the five steps of our knowledge modeling approach. Note, ontologies and FCA both use ‘concepts’ as part of their terminologies. In order to avoid ambiguity between these terms, we refer to FCA concepts as “formal concept” and concepts used in ontology designs as “class”.

4.4.1 Step 1: Vulnerabilities Information Acquisition and Pruning

Since the objective of this research is to identify and model information relevant to software security vulnerabilities; we conducted another survey of publicly (and free of charge) available SVDBs that model software vulnerabilities from different vendors and systems. The SVDBs were identified through an Internet search using the following two keywords, ‘vulnerability’ and ‘database’. We then manually inspected the top 100 search results returned by the search engine, to verify that they actually reference available SVDBs and to eliminate duplicate results. After the analysis of the search results we found 11 SVDBs that met our selection criteria. The selection criteria were: (1) the database host software known vulnerabilities, and (2) the vulnerability entry in the database is in the English language.

An overview of these 11 SVDBs is provided in Table 13 and Table 14, which include general statistics of these SVDBs and report on the vulnerability identification scheme used by these SVDBs.

While all surveyed SVDBs use some form of a vulnerability ID generation scheme, only the NVD dataset (Table 14 - D1) relies on a standardized identifier ID format (CVE-ID). The remaining SVDBs generate their own proprietary vulnerability IDs, with some SVDBs including the CVE-ID in their proprietary vulnerability ID generation.

Some SVDBs list affected products using the global standard naming scheme CPE, as shown in D1 and D9 where both use CPE’s well-formed name (WFN) (e.g., *wfn:part="a", vendor="Microsoft", product="internet_explorer", version="8.0", update="beta"*). The remaining SVDBs rely on unstructured text descriptions to identify and describe products affected by a vulnerability.

Table 13: 11 Security vulnerability databases

ID#	Name	Maintainer	URL	# archived entries
D1	National Vulnerabilities Database (NVD)	NIST	https://nvd.nist.gov/	94,657
D2	Exploit Database (ED)	Offensive Security	https://www.exploit-db.com/	38,415
D3	SecurityFocus (SF)	Security Focus	http://www.securityfocus.com/bid	95,460
D4	Vulnerability Notes Database (VND)	CERT/CC	http://www.kb.cert.org/vuls	94,735
D5	World Laboratory of Bugtraq (WLB)	CXSecurity	https://cxsecurity.com/wlb/about/	2,839
D6	Packet Storm Security (PSS)	Packet Strom	https://packetstormsecurity.com/	N/A*
D7	Vulnerability Lab (VL)	Evolution Security GmbH	https://www.vulnerability-lab.com/	882
D8	rapid7	Rapid7	https://www.rapid7.com/db/vulnerabilities	121,128
D9	VulDB	Scip AG	https://vuldb.com/	109,956
D10	Skybox Vulnerability Database (SVD)	Skybox Security	https://www.vulnerabilitycenter.com/#home	73,838
D11	Snyk Vulnerability DB	Snyk	https://snyk.io/vuln?packageManager=all	3,684

* The database presents the vulnerabilities information as HTML pages on the website, and does not provide a downloadable link to their archived entries.

Table 14: Vulnerabilities databases ID schemas and standards usages

ID#	Vulnerability ID scheme	CVE	CWE	CVSS	CPE
D1	CVE-{YYYY}-{NNNN...} (4 digit year, Variable length arbitrary digits)	yes	yes	yes	yes
D2	EDB-ID:{NNNNN} (5 fixed digits)	yes	no	no	no
D3	NNNNN {5 fixed digits}	yes	no	no	no
D4	VU#{NNNNNN} (6 fixed digits)	yes	yes	yes	no
D5	WLB-{YYYYMMNNNN} (4 digit year, 2 digit month, 4 fixed digits	yes	yes	no	no
D6	{NNNNNN} (6 fixed digits)	yes	no	no	no
D7	VL-{NNNN} (4 digits)	yes	no	yes	no
D8	{SSSS...} (Variable length arbitrary strings)	yes	no	yes	no
D9	{NNNNNN} (6 fixed digits)	yes	yes	yes	yes
D10	VUL={NNNN…} (Variable length of digits)	yes	no	yes	no
D11	SNYK-{L-P-NNNN..} (Programming language - product name -variable length digits)	yes	yes	no	no

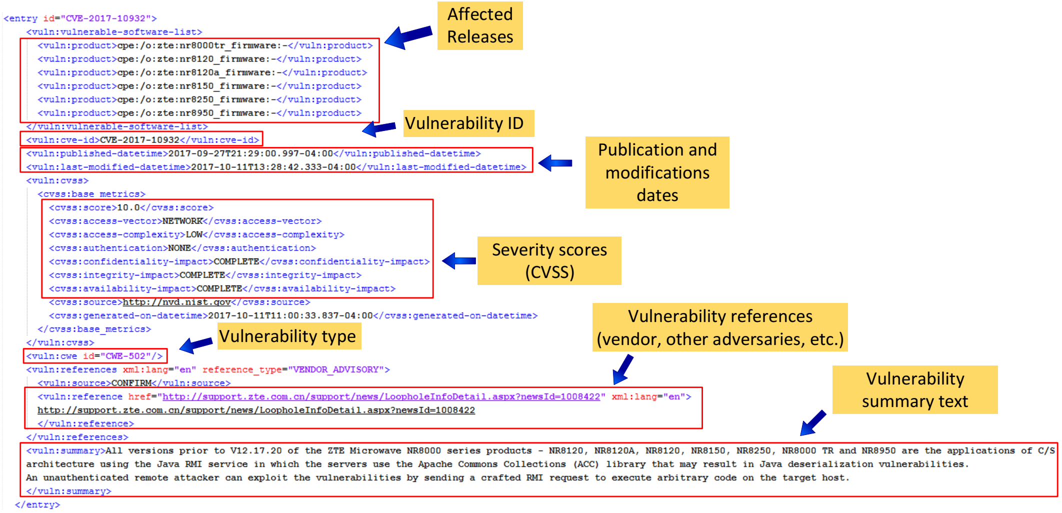
A commonality of the surveyed SVDBs is that they include the release date of a vulnerability. D7 also provides a popularity indicator for disclosed vulnerabilities (in terms of number of views), and 6 out of the 11 SVDBs include information about the person/organization who discovered a vulnerability. Also six SVDBs (D1, D4, D7, D8, D9, and D10) use the vulnerability scoring system (CVSS) to indicate the criticality of a reported vulnerability, while three other SVDBs rely on their own proprietary benchmarks (D5, D7, and D11).

Among other information provided by SVDBs are links to detailed vulnerability descriptions (D2, D3, D5, D7, D8, and D9), and status indicators (D2 and D7) showing if a vulnerability has been addressed and fixed (e.g., a software update availability). Mitigation measurements are provided by D2, D3, and D7-D10. All reviewed SVDBs include cross-references to entries in other SVDBs as well as links to software vendors describing a vulnerability. Descriptions to resolve the vulnerability are often references to patches provided by the vendor of the vulnerable software.

Accessibility to these public SVDBs and their vulnerability data is provided through web interfaces, with some of these SVDBs also including XML feeds (e.g., D1). Most of the databases update their vulnerabilities information on a daily basis, except for D3 and D4 which provide only weekly/monthly updates.

4.4.2 Step 2: Initial System-Specific Ontologies

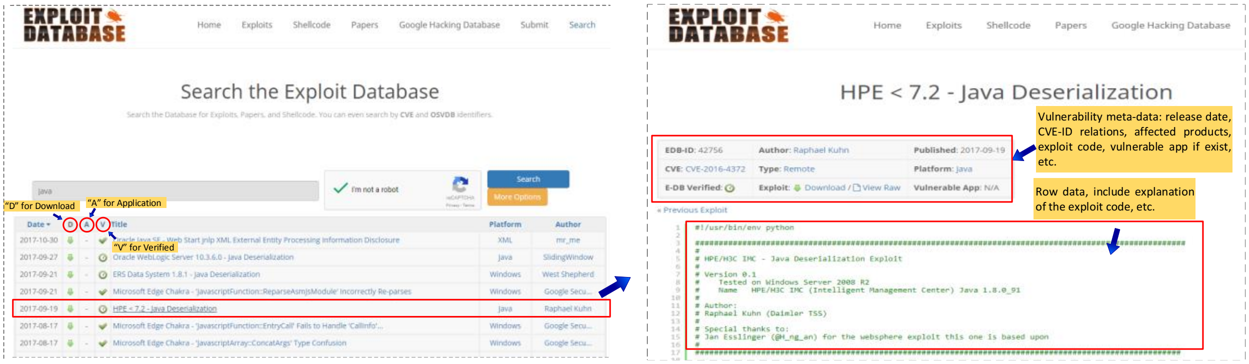
Next, we apply different types of manual data pre-processing steps to identify and extract concepts and their attribute definitions from the raw data extracted from the 11 SVDBs. A main challenge we had to deal with was the heterogeneity and ambiguity of the SVDBs data in terms of the underlying data models. Figure 14 (NVD - D1) and Figure 15 (ED - D2) illustrate examples of such schemata and information representation differences.



```
<entry id="CVE-2017-10932">
  <vuln:vulnerable-software-list>
    <vuln:product>cpe:/o:zte:nr8000tr_firmware:-</vuln:product>
    <vuln:product>cpe:/o:zte:nr8120_firmware:-</vuln:product>
    <vuln:product>cpe:/o:zte:nr8120a_firmware:-</vuln:product>
    <vuln:product>cpe:/o:zte:nr8150_firmware:-</vuln:product>
    <vuln:product>cpe:/o:zte:nr8250_firmware:-</vuln:product>
    <vuln:product>cpe:/o:zte:nr8950_firmware:-</vuln:product>
  </vuln:vulnerable-software-list>
  <vuln:cve-id>CVE-2017-10932</vuln:cve-id>
  <vuln:published-datetime>2017-09-27T21:29:00.997-04:00</vuln:published-datetime>
  <vuln:last-modified-datetime>2017-10-11T13:28:42.333-04:00</vuln:last-modified-datetime>
  <vuln:cvss>
    <cvss:base metrics>
      <cvss:score>10.0</cvss:score>
      <cvss:access-vector>NETWORK</cvss:access-vector>
      <cvss:access-complexity>LOW</cvss:access-complexity>
      <cvss:authentication>NONE</cvss:authentication>
      <cvss:confidentiality-impact>COMPLETE</cvss:confidentiality-impact>
      <cvss:integrity-impact>COMPLETE</cvss:integrity-impact>
      <cvss:availability-impact>COMPLETE</cvss:availability-impact>
      <cvss:source>http://nvd.nist.gov</cvss:source>
      <cvss:generated-on-datetime>2017-10-11T11:00:33.837-04:00</cvss:generated-on-datetime>
    </cvss:base_metrics>
  </vuln:cvss>
  <vuln:cwe id="CWE-502"/>
  <vuln:references xml:lang="en" reference_type="VENDOR_ADVISORY">
    <vuln:source>CONFIRM</vuln:source>
    <vuln:reference href="http://support.zte.com.cn/support/news/LoopholeInfoDetail.aspx?newsId=1008422" xml:lang="en">
    http://support.zte.com.cn/support/news/LoopholeInfoDetail.aspx?newsId=1008422
    </vuln:reference>
  </vuln:references>
  <vuln:summary>All versions prior to V12.17.20 of the ZTE Microwave NR8000 series products - NR8120, NR8120A, NR8120, NR8150, NR8250, NR8000 TR and NR8950 are the applications of C/S
  architecture using the Java RMI service in which the servers use the Apache Commons Collections (ACC) library that may result in Java deserialization vulnerabilities.
  An unauthenticated remote attacker can exploit the vulnerabilities by sending a crafted RMI request to execute arbitrary code on the target host.
  </vuln:summary>
</entry>
```

Affected Releases
Vulnerability ID
Publication and modifications dates
Severity scores (CVSS)
Vulnerability references (vendor, other adversaries, etc.)
Vulnerability type
Vulnerability summary text

Figure 14: NVD vulnerability entry - CVE-2017-10932 attributes.



Search the Exploit Database

Search the Database for Exploits, Papers, and Shellcode. You can even search by CVE and OSVDB identifiers.

java

Date ▾	D	A	V	Title	Platform	Author
2017-10-30	⬇	-	✓	Oracle Java SE - Web Start jnlp XML External Entity Processing Information Disclosure	XML	mr_me
2017-09-27	⬇	-	◷	Oracle WebLogic Server 10.3.6.0 - Java Deserialization	Java	SlidingWindow
2017-09-21	⬇	-	◷	ERS Data System 1.8.1 - Java Deserialization	Windows	West Shepherd
2017-09-21	⬇	-	✓	Microsoft Edge Chakra - 'JavascriptFunction::ReparseAsmJsModule' Incorrectly Re-parses	Windows	Google Secu...
2017-09-19	⬇	-	◷	HPE < 7.2 - Java Deserialization	Java	Raphael Kuhn
2017-08-17	⬇	-	✓	Microsoft Edge Chakra - 'JavascriptFunction::EntryCall' Fails to Handle 'CallInfo'...	Windows	Google Secu...
2017-08-17	⬇	-	✓	Microsoft Edge Chakra - 'JavascriptArray::ConcatArgs' Type Confusion	Windows	Google Secu...

HPE < 7.2 - Java Deserialization

EDB-ID: 42756	Author: Raphael Kuhn	Published: 2017-09-19
CVE: CVE-2016-4372	Type: Remote	Platform: Java
E-DB Verified: ◷	Exploit: ⬇ Download / 🗋 View Raw	Vulnerable App: N/A

```
#!/usr/bin/env python

##########################################################################################
#
# HPE/H3C IMC - Java Deserialization Exploit
#
# Version 0.1
#    Tested on Windows Server 2008 R2
#    Name   HPE/H3C IMC (Intelligent Management Center) Java 1.8.0_91
#
# Author:
# Raphael Kuhn (Daimler TSS)
#
# Special thanks to:
# Jan Esslinger (@H_ng_an) for the websphere exploit this one is based upon
#
##########################################################################################
```

Vulnerability meta-data: release date, CVE-ID relations, affected products, exploit code, vulnerable app if exist, etc.
Row data, include explanation of the exploit code, etc.

Figure 15: Vulnerabilities entries' attributes and information from exploit-database website.

As part of this knowledge modeling step, we extracted system-specific ontologies for each SVDB using their schemata and documentation. Figure 16 shows an example of the two system-specific ontologies we extracted for D1 and D2.

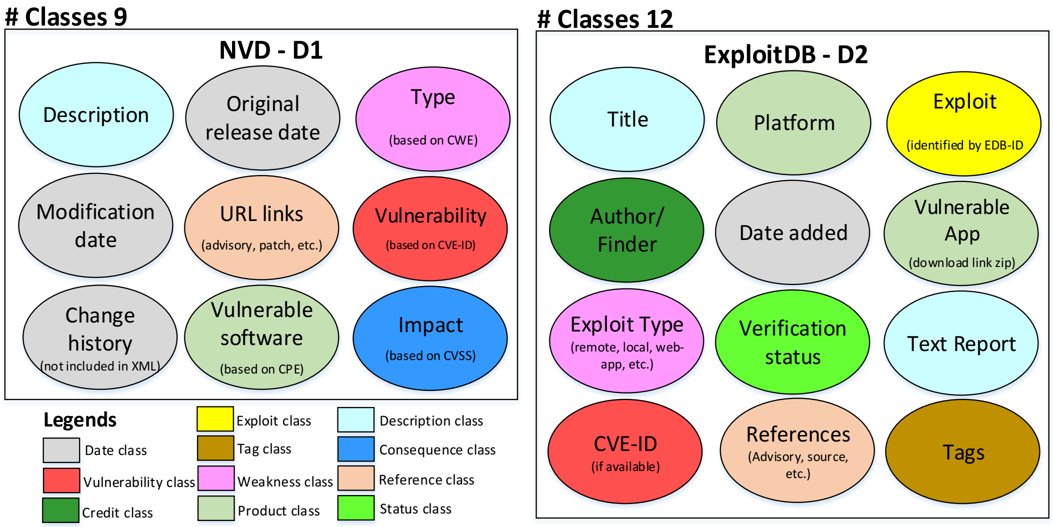


Figure 16: The main classes extracted from D1 (left) and D2 (right) to create their system-specific ontologies.

It should be noted that each class can include sub-classes that are extracted based on descriptions provided by the SVDB, such as *original release date* and *modification date,* which will become subclasses of the *date* class. Similarly, we extract properties for all classes/sub-classes which are used to describe the system-specific ontologies. In total, we extracted 131 classes and 201 properties from the 11 SVDBs (see Figure 17).

The quite large number of classes and properties are due to the fact that the same class or property may exist several times under different names in these SVDBs. For example, the representation of vulnerability modification date information varies across the SVDBs; in D1 and D8 it is referred to as *modified date*, whereas in D3 *update date*, in D4 *date last update*, and in D10 *last modified date* are used to capture modification date. In the following section, we describe the process which we applied to remove some of this ambiguity during the mapping of our system-level ontologies.

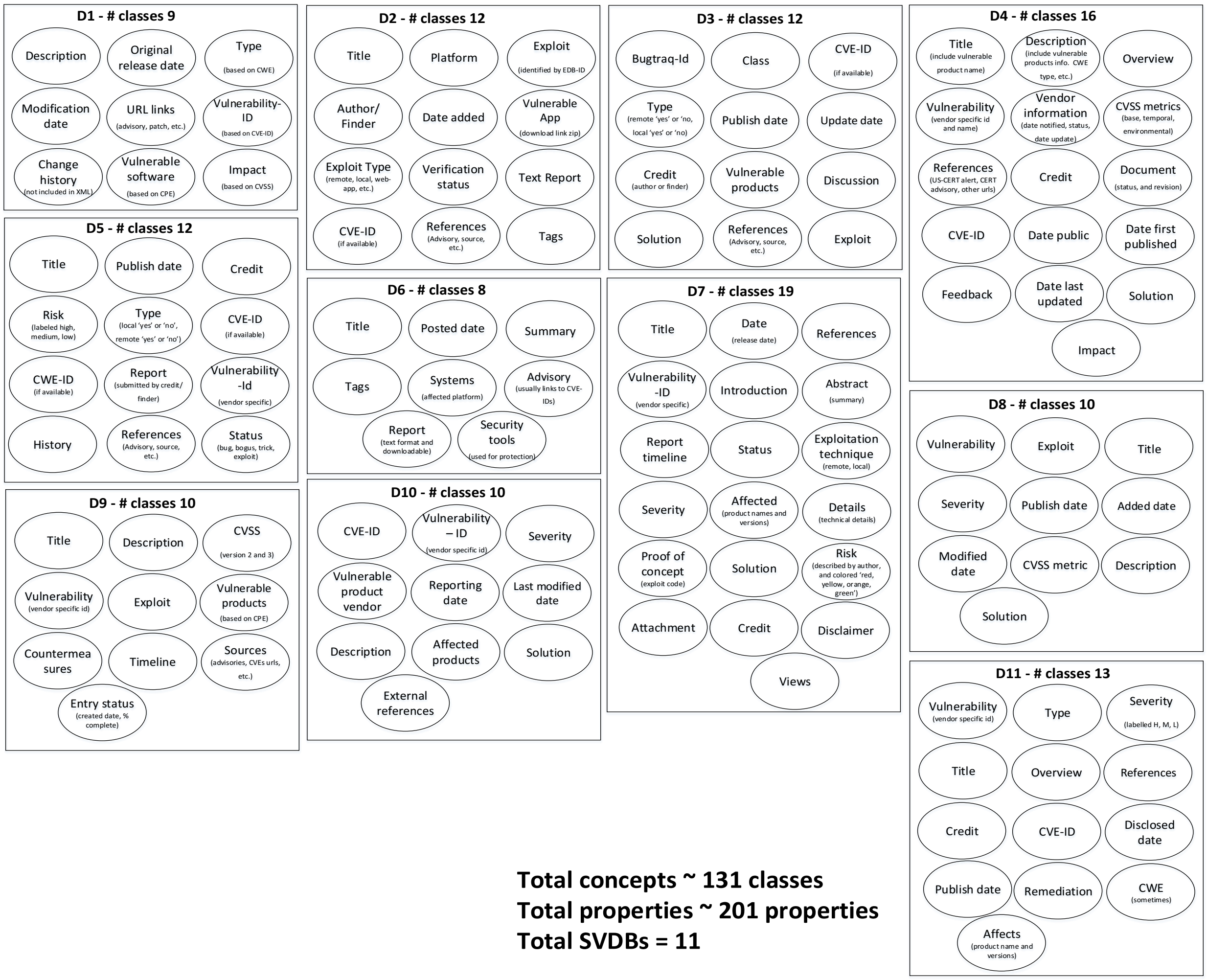

D1 - # classes 9
Description
Original release date
Type (based on CWE)
Modification date
URL links (advisory, patch, etc.)
Vulnerability-ID (based on CVE-ID)
Change history (not included in XML)
Vulnerable software (based on CPE)
Impact (based on CVSS)
D2 - # classes 12
Title
Platform
Exploit (identified by EDB-ID
Author/ Finder
Date added
Vulnerable App (download link zip)
Exploit Type (remote, local, web-app, etc.)
Verification status
Text Report
CVE-ID (if available)
References (Advisory, source, etc.)
Tags
D3 - # classes 12
Bugtraq-Id
Class
CVE-ID (if available)
Type (remote 'yes' or 'no, local 'yes' or 'no)
Publish date
Update date
Credit (author or finder)
Vulnerable products
Discussion
Solution
References (Advisory, source, etc.)
Exploit
D4 - # classes 16
Title (include vulnerable product name)
Description (include vulnerable products info. CWE type, etc.)
Overview
Vulnerability (vendor specific id and name)
Vendor information (date notified, status, date update)
CVSS metrics (base, temporal, environmental)
References (US-CERT alert, CERT advisory, other urls)
Credit
Document (status, and revision)
CVE-ID
Date public
Date first published
Feedback
Date last updated
Solution
Impact
D5 - # classes 12
Title
Publish date
Credit
Risk (labeled high, medium, low)
Type (local 'yes' or 'no', remote 'yes' or 'no')
CVE-ID (if available)
CWE-ID (if available)
Report (submitted by credit/ finder)
Vulnerability -Id (vendor specific)
History
References (Advisory, source, etc.)
Status (bug, bogus, trick, exploit)
D6 - # classes 8
Title
Posted date
Summary
Tags
Systems (affected platform)
Advisory (usually links to CVE-IDs)
Report (text format and downloadable)
Security tools (used for protection)
D7 - # classes 19
Title
Date (release date)
References
Vulnerability -ID (vendor specific)
Introduction
Abstract (summary)
Report timeline
Status
Exploitation technique (remote, local)
Severity
Affected (product names and versions)
Details (technical details)
Proof of concept (exploit code)
Solution
Risk (described by author, and colored 'red, yellow, orange, green')
Attachment
Credit
Disclaimer
Views
D8 - # classes 10
Vulnerability
Exploit
Title
Severity
Publish date
Added date
Modified date
CVSS metric
Description
Solution
D9 - # classes 10
Title
Description
CVSS (version 2 and 3)
Vulnerability (vendor specific id)
Exploit
Vulnerable products (based on CPE)
Countermea sures
Timeline
Sources (advisories, CVEs urls, etc.)
Entry status (created date, % complete)
D10 - # classes 10
CVE-ID
Vulnerability – ID (vendor specific id)
Severity
Vulnerable product vendor
Reporting date
Last modified date
Description
Affected products
Solution
External references
D11 - # classes 13
Vulnerability (vendor specific id)
Type
Severity (labelled H, M, L)
Title
Overview
References
Credit
CVE-ID
Disclosed date
Publish date
Remediation
CWE (sometimes)
Affects (product name and versions)
Total concepts ~ 131 classes
Total properties ~ 201 properties
Total SVDBs = 11

Figure 17: Manually identified vulnerabilities concepts from 11 SVDBs.

4.4.3 Step 3: Ontology Mapping Using FCA

In this step of our knowledge modeling approach we used the extracted system level ontologies data from the previous modeling step to create and update our initial software vulnerability domain ontology. More specifically, we focus on classifying any attribute (e.g., concept, property, or relation) that can be promoted in our knowledge model from the *System* to the *Domain* layer. The *Domain* layer not only promotes such reuse of concepts across system level ontologies, but also improves traceability among system level ontologies by unifying the overall knowledge representation. In our modeling approach we take advantage of FCA for identifying potential domain concepts using (1) context formation and (2) context composition. For the context formation we use the vulnerability information of the system-specific ontologies as input to generate a one valued context table for each SVDB. These generated contexts are then combined during the context composition step to create a new merged formal context table for all SVDBs. In what follows, we describe each of these two processing steps using our dataset D1 and D2 (from Table 13) as an illustrative example.

Context formation: The input data for the FCA algorithm is a cross-table, which describes the relationships between objects (represented by table rows—in our case, the SVDBs data sources) and attributes (represented by table columns—in our case, all elements from the system-specific ontology). For example, in Table 15 the "x" for K1.A8 indicates that data source D1 uses CVE-ID in its data model.

Table 15: D1 context table K1

Context	K1.A1	K1.A2	K1.A3	K1.A4	K1.A5	K1.A6	K1.A7	K1.A8	K1.A9
D1	*x*	*x*	*x*	*X*	*x*	*x*	*x*	*X*	*x*

K1.A1: Original release date	K1.A2: Modified date	K1.A3: Change history
K1.A4: Description summary	K1.A5: URL links	K1.A6: Uses CPE
K1.A7: Uses CWE	K1.A8: Uses CVE-ID	K1.A9: Uses CVSS

Table 16: D2 context table K2

Context	K2.A1	K2.A2	K2.A3	K2.A4	K2.A5	K2.A6	K2.A7	K2.A8	K2.A9	K2.A10	K2.A11	K2.A12
D2	*x*	*x*	*x*	*x*	*x*	*X*	*x*	*x*	*x*	*x*	*x*	*x*

K2.A1: Title	K2.A2: Platform	K2.A3: Exploit – vendor-id	K2.A4: Author
K2.A5: Date added	K2.A6: Vulnerable app link	K2.A7: Exploit type – 'remote, local'	K2.A8: Verification status
K2.A9: Text report	K2.A10: Related CVE-ID	K2.A11: References	K2.A12: Tags

Context Composition: Given the two contexts $K_1 := (O_1, A_1, R_1)$ and $K_2 := (O_2, A_2, R_2)$, the integrated context $K \coloneqq (O, A, R)$ is computed by performing a *disjoint union* of object sets of the two contexts:

$$O = O_1 \cup^* O_2 \qquad (1)$$

A and R are assigned A_1 and R_1 from K_1 at this stage, i.e., A = A_1 and R = R_1. Table 17 shows the context K after the above operations.

Table 17: The context table after object union operation

Context	K1.A1	K1.A2	K1.A3	K1.A4	K1.A5	K1.A6	K1.A7	K1.A8	K1.A9
D1	*x*	*x*	*x*	*X*	*x*	*x*	*x*	*X*	*x*
D2									

K1.A1: Original release date K1.A2: Modified date K1.A3: Change history
K1.A4: Description summary K1.A5: URL links K1.A6: Uses CPE
K1.A7: Uses CWE K1.A8: Uses CVE-ID K1.A9: Uses CVSS

For the mapping process we introduce three integration rules to guide the domain expert while incrementally combining the individual system-level context tables into our new merged context table for all 11 SVDBs.

Rule #1 Identical attributes: If $a_i \in A_1$ AND $a_j \in A_2$ AND $a_i \equiv a_j$. For example, *date added* (K2.A5 in K2—see Table 16) is identical in its semantic meaning to the *original release date* (K1.A1 in K1—see Table 15). By applying rule #1, both attributes will be merged under a new common name (i.e., label) *entry_release_date* (as shown in Table 18—K.A1 in K) which has a relation (*x*) for both (D1 and D2) in the context table K. Table 18 displays the results after applying rule #1, showing the unified attributes K1.A1 and K1.A5 from context K1, and K2.A5 and K2.A6 from context K2, which become K.A1 and K.A5 in the merged context K.

Table 18: The context table K after an equivalent match

Context	**K.A1**	K1.A2	K1.A3	K1.A4	**K.A5**	K1.A6	K1.A7	K1.A8	K1.A9
D1	***x***	*x*	*x*	*x*	***x***	*x*	*x*	*x*	*x*
D2	***x***				***x***				

K.A1: entry_release_date K1.A2: Modified date K1.A3: Change history
K1.A4: Description summary **K.A5: entry_URL links** K1.A6: Uses CPE
K1.A7: Uses CWE K1.A8: Uses CVE-ID K1.A9: Uses CVSS

It should be noted these new attributes are labeled with *entry_* (as shown in Figure 18) followed by the actual attribute name.

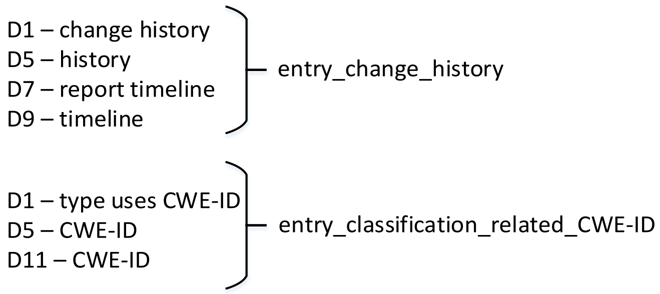


Figure 18: Example of unifying the attributes names from the system-specific data sources.

Rule #2 Attribute subsumption hierarchies: If ($a_i \in A_1$ AND $a_j \in A_2$) AND ($a_i \supset a_j$ OR $a_i \subset a_j$). For example, *Description summary* (K1.A4 in K1) is a super-class of *Text report* (K2.A9 in K2). By applying rule #2, the two attributes will be included in the combined context table K, *entry_description_summary* as a common attribute name (K.A4) and *entry_text_report* (K.A10) reflecting the extension used in K2. The updated context table K (Table 19) after applying this rule is as follows:

Table 19: The context table after the second mapping type

Context	**K.A1**	K1.A2	K1.A3	**K.A4**	**K.A5**	K1.A6	K1.A7	K1.A8	K1.A9	**K.A10**
D1	***x***	*x*	*x*	***x***	***x***	*x*	*x*	*x*	*x*	
D2	***x***			***x***	***x***					***x***

K.A1: entry_release_date K1.A2: Modified date K1.A3: Change history
K.A4: entry_description summary **K.A5: entry_URL links** K1.A6: Uses CPE
K1K1.A7: Uses CWE K1.A8: Uses CVE-ID K1.A9: Uses CVSS
K.A10: entry_text_report

Rule #3 Unique attributes. If $a_i \in A_1$ AND $a_j \in A_2$ AND $a_i \not\equiv a_j$. In the case that neither rule #1 nor rule #2 are applicable, then both attribute instances will be automatically inserted into the combined context table K (Table 20).

Table 20: The context table after the third mapping type

Context	**K.A1**	K1.A2	K1.A3	**K.A4**	**K.A5**	K1.A6	K1.A7	K1.A8	K1.A9	**K.A10**	**K2.A11**	**K2.A12**	…
D1	***x***	*x*	*x*	***x***	***x***	*x*	*x*	*x*	*x*				
D2	***x***			***x***	***x***					***x***	***x***	***x***	...

K.A1: entry_release_date K1.A2: Modified date K1.A3: Change history
K.A4: entry_description summary **K.A5: entry_URL links** K1.A6: Uses CPE
K1.A7: Uses CWE K1.A8: Uses CVE-ID K1.A9: Uses CVSS
K.A10: entry_text_report K2.A11: Title K2.A12: Platform

This incremental mapping between individual context tables and the combined context table continues until a complete merged context table for all 11 SVDBs is created (see Table 21). As a result of this integration process, we obtain a final context table with 23 attributes of which 16 are shared among different SVDBs and 7 (K7.A4, K2.A5, K1.A8, K3.A12, K6.A16, K7.A25, and K4.A28) are unique by being only used in individual SVDBs.

Table 21: Final context table K

	K.A1	**K.A2**	**K.A3**	K7.A4	K2.A5	**K.A6**	**K.A7**	K1.A8	**K.A9**	**K.A10**	**K.A11**	K3.A12	**K.A13**	**K.A14**	**K.A15**
D1	x	x	x					x	x		x				x
D2					x	x							x		x
D3						x					x	x	x	x	x
D4			x			x	x		x	x	x		x		x
D5	x	x				x				x			x	x	x
D6			x												x
D7	x		x	x		x				x				x	
D8			x				x		x	x	x			x	
D9	x		x						x					x	x
D10			x							x	x		x		x
D11		x	x			x	x			x			x		x

K.A1: entry_change_history
K.A2: entry_classification_related_CWE-ID
K.A3: entry_description_summary
K7.A4: disclaimer
K2.A5: exploit_vendor_specific_id

K.A6: entry_finder
K.A7: entry_first_release_date
K.1A8: id_uses_CVE-ID
K.A9: entry_impact _uses_CVSS
K.A10: entry_impact_vendor_specific

K.A11: entry_modified_date
K3.A12: related_bugtraq-id
K.A13: entry_related_CVE-ID
K.A14: entry_related_exploit
K.A15: entry_related_external_url_links

	K6.A16	**K.A17**	**K.A18**	**K.A19**	**K.A20**	**K.A21**	**K.A22**	**K.A23**	**K.A24**	K7.A25	**K.A26**	**K.A27**	K4.A28	**K.A29**	**K.A30**
D1		x													x
D2		x		x	x	x	x		x			x		x	
D3		x	x					x	x					x	
D4		x	x			x	x	x			x		x		
D5		x				x	x		x		x				
D6	x	x			x	x	x					x		x	
D7		x	x	x		x	x		x	x	x			x	
D8		x	x			x					x				
D9		x	x	x		x					x				x
D10		x	x								x		x	x	
D11		x	x			x			x		x			x	

K6.A16: related_security_tools
K.A17: entry_release_date
K.A18: entry_solution
K.A19: entry_status
K.A20: entry_tags

K.A21: entry_title
K.A22: entry_unstructured_text_report
K.A23: entry_users_feedback
K.A24: entry_vendor_specific_classification
K7.A25: views

K.A26: entry_vulnerability_vendor_specific_id_name
K.A27: entry_vulnerable_platform
K4.A28: vulnerable_product_vendor_information
K.A29: entry_vulnerable_products
K.A30: entry_vulnerable_products_uses_CPE

4.4.4 Step 4: Establishing Concept Hierarchy Using FCA Lattice

After having established our merged context table K for all SVDBs, we can now create an FCA lattice for Table 21. This lattice can provide knowledge engineers with some additional visual guidance while classifying concepts into system and domain level concepts (see Figure 19).

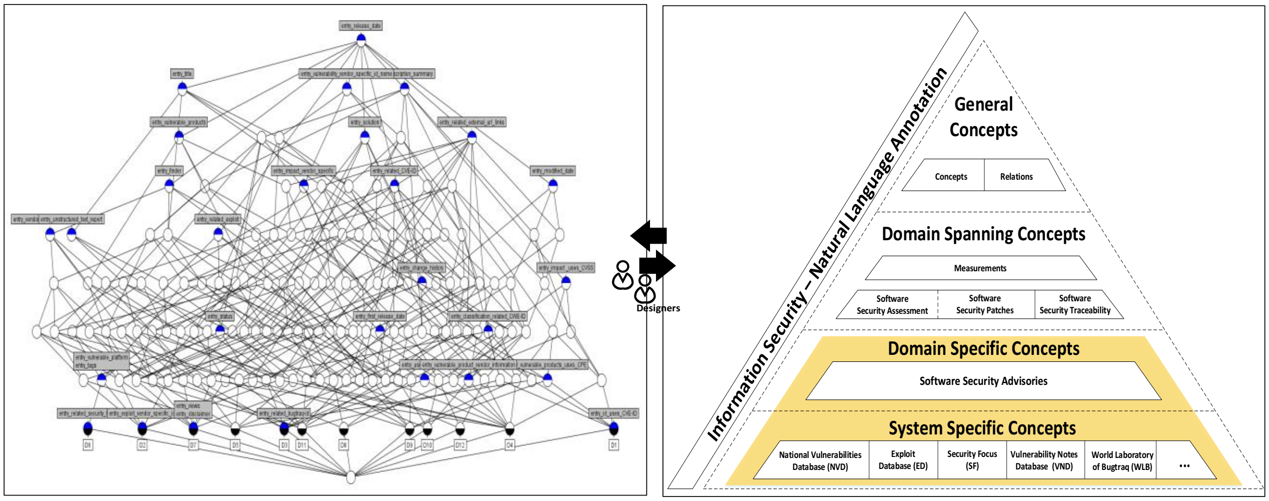

Designers
Information Security – Natural Language Annotation
General Concepts
Concepts
Relations
Domain Spanning Concepts
Measurements
Software Security Assessment
Software Security Patches
Software Security Traceability
Domain Specific Concepts
Software Security Advisories
System Specific Concepts
National Vulnerabilities Database (NVD)
Exploit Database (ED)
Security Focus (SF)
Vulnerability Notes Database (VND)
World Laboratory of Bugtraq (WLB)
…

Figure 19: Concept lattice for the merged context table to guide classification of concepts in domain and system level concepts.

The main challenge for this modeling step is to determine what constitutes a good domain concept/attribute. Ideally any domain concept/attribute should be shared among all system level ontologies, therefore providing maximum reuse of this concept and improving traceability among the system level ontologies. However, when dealing with multiple system ontologies, it becomes increasingly difficult to identify such concepts. We therefore relax our original objective of a domain concept by allowing a concept to be promoted from a system level concept to the domain level concept if it: (a) captures the core of the domain of discourse, (b) is a formal concept that is further extended by other concepts, and (c) is a formal concept in the FCA lattice that is shared by other formal concepts. In our modeling approach, we therefore introduce a semi-automated approach which (1) uses a stability measure to identify potential domain level concepts, and (2) a domain-expert who makes the final modeling decision by determining which concept should be promoted to the domain level or should remain at the system level.

The FCA stability measure assesses how close a concept's intent depends on other objects it extents. In other words, it reflects the probability of preserving its intent after removing an arbitrary number of objects. Thus, a high stability value indicates that a concept represents a cohesive set of resources or, equally, it can represent a concept that occurs commonly in a domain. The stability measures we apply were first introduced by [216], and later extended to formal concepts [217]. We use Equation (2) to calculate the stability of a given formal concept, according to the definition in Kuznetsov et al. [217], as shown. Given a concept C, concept stability $\mathrm{Stab(C)}$ is defined as

$$\mathrm{Stab(C)} = \frac{|\{\mathrm{s} \in \wp(\mathrm{Ext(C)}) \mid \mathrm{s}\prime = \mathrm{Int(C)}\}|}{2^{|\mathrm{Ext(C)}|}} \qquad (2)$$

Where the relative number of subsets of the concept extent (denoted by $\mathrm{Ext(C)}$), whose description (i.e., the result of $(.)'$) is equal to the concept intent (denoted by $\mathrm{Int(C)}$) where $\wp(P)$ is the power set of P.

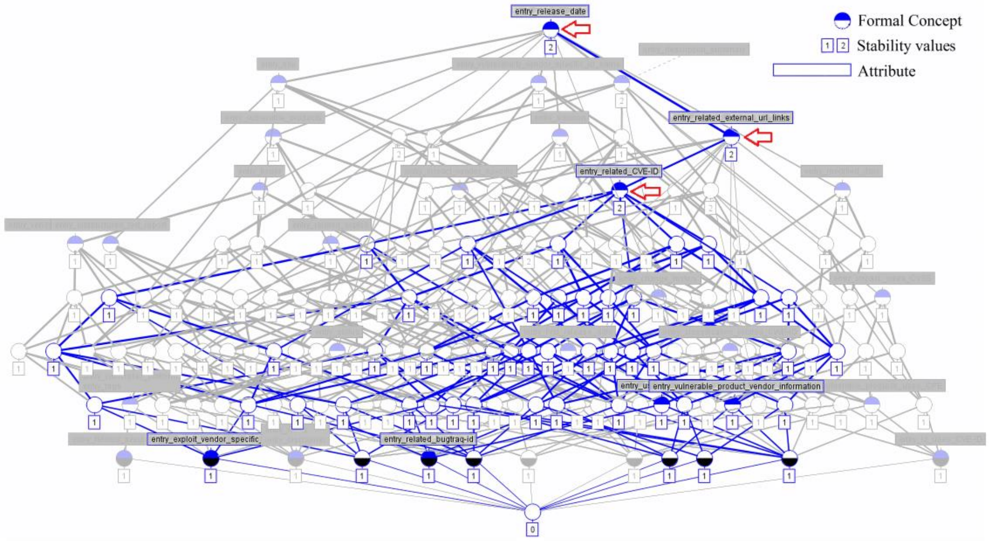


Figure 20: Stability measure applied on the combined concept lattice.

For generating the FCA lattice and the computation of the stability measure we use the ConExp[58] tool for our SVDBs context table (Table 21). Figure 20 shows the lattice graph and the stability values produced by the tool. The stability measure values are 1 (minimum) and 2 (maximum), with only 7 out the 159 formal concepts having the maximum stability value of 2. We use a stability measure threshold to promote a system level to a domain level concept. The domain expert will then decide whether one of these seven formal concepts will be mapped to a

[58] ConExp tool at http://www.sf.net/projects/conexp/

new domain concept in our domain ontology or become an attribute of an existing domain concept (object or data property).

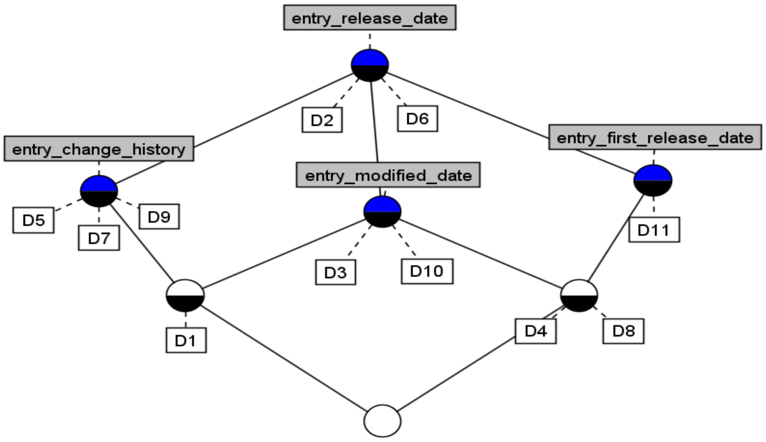


Figure 21: Vulnerability related dates concepts and sub-concepts.

Example: Why is the ***entry_release_date*** attribute promoted (based on the stability measure) to the domain-specific layer as a property for the vulnerabilities' *Date* class?

As shown in Figure 21, the *entry_release_date* is shared among all databases creating the formal concept ({D1,…, D11},{*entry_release_date*}). At the same time it has the highest stability value when compared with other date attributes in the lattice (Figure 21, e.g., *entry_change_history*, *entry_modified_date*, and *entry_first_release_date*). However, *entry_release_date* will be promoted to the upper levels (domain layer), and the other attributes *entry_change_history*, *entry_modified_date*, and *entry_first_release_date* will remain in the system-specific layer.

4.4.5 Step 5: A Unified Knowledge Representation

The last step of our approach is to instantiate our knowledge model, with each of its layers corresponding to a separate ontology artifact (OWL file). Table 22 provides an overview of key statistics of these layers, including the number of axioms (*#of axioms)*, number of classes (*#of classes)*, number of object properties (*#of object-properties)*, and number of data properties (*# data-properties*) found in each abstraction layer.

Table 22: Ontologies artifacts metrics

Layer	Ontology Name	# Axioms	# Classes	# Object Properties	# Data Properties
General	main.owl	130	10	14	11
Domain-spanning	vulnerabilitis.owl	41	17	6	0
	measurement.owl	55	10	5	0
Domain-specific	securityDB.owl	176	28	18	19
System-specific	securityDB-nvd.owl	41	9	0	9
	securityDB-exploitdb.owl	26	5	2	8
	securityDB-SF.owl	4	5	0	0
	securityDB-VND.owl	7	5	2	0
	securityDB-WLB.owl	6	3	0	0
	securityDB-PSS.owl	12	4	0	3
	securityDB-VL.owl	15	7	2	1
	securityDB-rapid7.owl	2	2	0	0
	securityDB-VulDB.owl	2	2	0	0
	securityDB-SVD.owl	0	0	0	0
	securityDB-SnykVDB.owl	8	4	0	2
Total		**525**	**111**	**49**	**53**

Ontologies at the system level can now extend classes from the upper layers either directly (*Domain* layer) or indirectly (from the *Domain-spanning* and *General* layers). For example, D1 (*securityDB-nvd.owl*) can now extend higher-level classes such as the domain-specific (*securityDB.owl*), domain-spanning (*vulnerabilities.owl* and *measurement.owl*), and the general layers (*main.owl*). In some cases (e.g., D10 - securityDB-SVD.owl) all vulnerability concepts are already covered by the domain-specific ontology and no system-specific ontology extensions are needed; the *securityDB-SVD.owl* will be empty.

Given our ontological modeling approach and the support for inference services, we can now use rules (see Listing 1) to infer missing information (through implicit relations). For example, given an exploit attack published in D2[59] which states:

"*The Enterprise version of SyncBreeze is affected by a Remote Denial of Service vulnerability. The web server does not check bounds when reading server request in the Host header on making a connection, resulting in a classic Buffer Overflow that causes a Denial of Service. To exploit the vulnerability only is needed use the version 1.1 of the HTTP protocol to interact with the application*".

While most of the relevant vulnerability information is provided, other information such as severity score, attack impact, and attack classification are not mentioned in the D2 record for this

[59] https://www.exploit-db.com/exploits/43344/

attack. However, D1[60] contains this additional vulnerability information (e.g., has attack classification, Buffer Errors (CWE-119), and vulnerability impact, CVSS score 7.5 out of 10 - high). Using the semantic rule in Listing 1, we can now infer the missing information in D2 from D1.

```
**IF** ($D_x$ has CVE-ID && CWE-ID && CVSS) **AND** ($D_y$ has CVE-ID) **THEN** $D_y$has CWE-ID, CVSS
**IF** ($D_x$ has CVE-ID && Author) **AND** ($D_y$ has CVE-ID) **THEN** $D_y$ has Author
**IF** ($D_y$ has CVE-ID && exploit code) **AND** ($D_x$ has CVE-ID) **THEN** $D_x$ has exploit code
```

Listing 1: Semantic rules to infer vulnerabilities representations standards.

4.5 Use Cases Scenarios

In this section, we illustrate the applicability of our knowledge model through examples that take advantage of our unified ontological representation. The results demonstrate that our knowledge model cannot only unify these heterogeneous vulnerability data-sources but can also enable new and more flexible types of vulnerability analysis.

Implementation: For both of our examples, we reuse again our previous NVD (D1) and Exploit-DB (D2) datasets. NVD is arguably the world's largest database of publicly known vulnerabilities in software systems [38], and Exploit-DB[61] is a CVE compliant archive of public exploits and software being affected by these exploits. Exploit-DB is mainly used by penetration testers and vulnerability researchers. Common to both data sources is that they can be considered to be information silos, with no unified representation that would allow for a seamless information exchange or exploration facilities across these two SVDBs. For example, when a CVE designation is assigned to a reported vulnerability in D1, details about this vulnerability are often not published until a technical review of the vulnerability is completed. Such a review typically also includes assigning a severity score and additional CWE classification information. On the other hand, D2 publishes the CVE-ID of a vulnerability together with its description and a proof of concept exploit. These information differences can not only result in situations where the same vulnerability (CVE-ID) might be published with a different reporting date, but also each SVDB provides different information details about the same vulnerability. In what follows, we show how our unified knowledge representation can eliminate the traditional information

[60] https://nvd.nist.gov/vuln/detail/CVE-2017-17088
[61] https://www.exploit-db.com/about-exploit-db/

silos these SVDBs have preserved and improve the accessibility and traceability of information among these SVDBs.

Data collection, extraction, and population: For the data collection, we downloaded and parsed all available XML data feeds from D1 and populated them in our D1-system-specific ontology. For the D2 dataset, we implemented a Java script to scrape D2's website to extract the meta-data for each published exploit (CVE-IDs if exist, author, links, etc.) and populated the downloaded facts in our D2-system-specific ontology. It should be noted that for all of our system- and domain-specific ontologies we use the same URI generator to create unique, dereferenceable URIs for each fact in our knowledge base (Figure 22 shows the URIs namespaces for each ontology and the nomenclature used).

```
owl: <http://www.w3.org/2002/07/owl#>
rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
main: <http://se-on.org/ontologies/general/2012/02/main.owl#>
sevont: <http://aseg.cs.concordia.ca/segps/ontologies/domain-specific/2015/02/securityDBs.owl#>
exploitdb:<http://aseg.cs.concordia.ca/segps/ontologies/domain-specific/2017/01/securityDBs-exploitdb.owl#>
nvd: <http://aseg.cs.concordia.ca/segps/ontologies/domain-specific/2015/02/securityDBs-nvd.owl#>
```

Figure 22: Overview of namespaces and nomenclature used.

The results of our fact extraction and population step are two system-ontologies, which consist of 94,657 vulnerabilities (D1) and 38,415 exploit entries (D2) ; see Table 23.

Table 23: Dataset statistics

ID	**Name**	**Total entries**	**# triples**
D1	National Vulnerabilities Database (NVD)	94,657	1,793,393,3
D2	Exploit Database (ED)	38,415	453,236

Next, we illustrate how our modeling approach can benefit SVDB users by: (1) identifying inconsistency among disclosure dates of vulnerabilities, and (2) enriching SVDB system level ontologies with missing vulnerability information from other SVDBs.

4.5.1 Use Case Scenario #1: Identifying Inconsistencies in Vulnerability Public Disclosure Dates

Determining the public disclosure date of a vulnerability is vital to understanding the life cycle of a vulnerability. Previous studies (e.g., [102], [167], [168], [106], and [169]) have relied in their analysis on only a single SVDB (e.g., NVD) as their information source. While the CVE publication date used in these studies corresponds to the publication date in NVD, it is not necessarily the date the vulnerability was publicly disclosed [218]. In our approach we compare the publication dates of vulnerabilities in different SVDBs to identify inconsistencies among the dates and to provide security engineers with more accurate reporting date results.

Results and discussion: For our analysis, we take advantage of our ontological representation which allows us to create a simple SPARQL query (Listing 2) that links all exploits based on their CVE-IDs from the two databases (D1 and D2) and returns the release date for each vulnerability recorded in the individual dataset.

```
PREFIX[…]
SELECT DISTINCT ?exploit ?expDate ?vulnerability ?vulDate
WHERE{
    ?exploit rdf:type sevont:Exploit.
    ?vulnerability rdf:type sevont:Vulnerability.
    ?exploit sevont:hasEntryReleaseDate ?expDate.
    ?vulnerability sevont:hasEntryReleaseDate ?vulDate.
    ?vulnerability sevont:hasExploit ?exploit.
}
```

Listing 2: Query to retrieve publish (Release) dates between vulnerabilities (D1)

The query results show that 21,654 CVEs appear in both datasets and that of these 16,337 (75%) are disclosed in D2 prior to being published in D1 (with a median number of seven days between the two reports). On the other hand, 3,848 (18%) of vulnerabilities were published first in D1 before they were included in D2. Only 7% of the vulnerabilities show the same publication date in both SVDBs. Table 24 shows an example of the retrieved data from our knowledge base for the vulnerabilities publication dates for D1 and D2.

Table 24: Example information retrieved from our Knowledge Base for NVD and Exploit-DB as denoted in Query 1

Exploit-DB entry	Release Date	NVD entry	Release Date
Exploit-DB: 43376	20/12/2017	NVD: CVE-2017-17692	21/12/2017
Exploit-DB: 43363	14/12/2017	NVD: CVE-2017-17411	21/12/2017
Exploit-DB: 43378	20/12/2017	NVD: CVE-2017-17752	20/12/2017
Exploit-DB: 18818	01/05/2012	NVD: CVE-2012-2576	20/12/2017
Exploit-DB: 43377	20/12/2017	NVD: CVE-2017-17759	19/12/2017
Exploit-DB: 43355	18/12/2017	NVD: CVE-2017-15048	19/12/2017
Exploit-DB: 43354	18/12/2017	NVD: CVE-2017-15049	19/12/2017
Exploit-DB: 43344	15/12/2017	NVD: CVE-2017-17088	19/12/2017

For example, CVE-2012-2576 is a vulnerability in SolarWinds Storage manager 5.1.0 and earlier[62] that was first discussed and assigned a CVE-ID in D2 in May 2012. The same vulnerability was not published in D1 until December 2017. Our results show that users who rely only on D1 and are unaware of exploits reported in D2 will not receive any alerts from D1 until December 2017. However, once a vulnerability becomes public (in this case May 2012), its likelihood for being exploited increases significantly, since the vulnerability is now publicly announced.

4.5.2 Use Case Scenario #2: Enriching Vulnerability Data

As we discussed earlier, SVDBs are a rich data source that can be used for mining and analysis of security issues and their fixes. However, the level of details, the completeness, and the quality of vulnerability information may differ significantly among SVDBs. For example, a security engineer relying only on D2 would have to rely on CVE-ID to identify a vulnerability. While CVE-IDs are widely used to describe reported vulnerabilities, alternative or supplementary information might exist in other SVDBs.

Results and discussion: Query 2, an extension of the SPARQL query (Listing 2), establishes a semantic link between D1 and D2 using the CVE-ID captured by both SVDBs. Given SPARQL query (Listing 3), we can now automatically infer missing vulnerability information in D2 such as the exploit type (described by CWE standard) and the exploit severity score (described by CVSS standard).

[62] https://www.offensive-security.com/0day/solarshell.txt

```
PREFIX[…]
SELECT DISTINCT ?exploit ?vulnerability ?weakness
                ?severityLevel ?CVSSscore
WHERE{
    ?exploit rdf:type sevont:Exploit.
    ?vulnerability rdf:type sevont:Vulnerability.
    ?vulnerability sevont:hasExploit ?exploit.
    ?vulnerability sevont:hasSeverity ?severityLevel.
    ?vulnerability sevont:hasWeakness ?weakness.
    ?vulnerability sevont:hasSeverityScore ?CVSSscore .
}
```

Listing 3: Query to retrieve the missing vulnerability information between vulnerabilities (D1) and related exploits (D2).

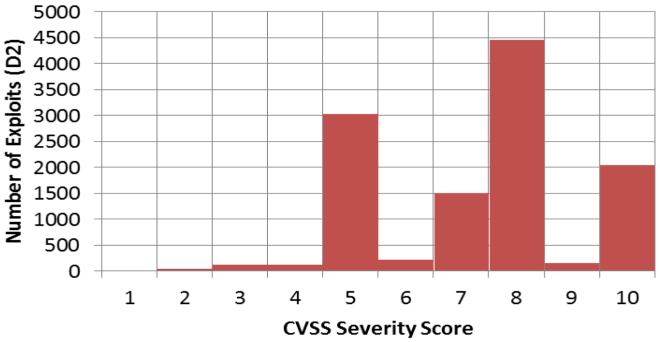


Figure 23: Distribution of CVSS severity scores, which are on a scale of 0 to 10, rounded to the nearest integer.

CVSS scores assigned to D2 exploits. The results from query (Listing 3) can now, for example, be used to classify exploits in D2 based on their CVSS severity scores. CVSS severity scores range from 0 to 10. Our query results (see Figure 23) show that vulnerabilities reported in D2 data consist mostly of vulnerabilities with a severity score of 5-10 (medium to high severity).

CWEs information assigned to D2 exploits. Based on Listing 3, we can now use the CWE standard (captured in D1) for an additional classification of software exploits in D2 using the CVE-ID link we established among the two datasets. The results of our query show that for 54% of exploits a CWE-ID can be identified and classified based on the CWE standard. Table 25 lists the most common software weaknesses, including the frequency with which they occur in D2. SQL injection, buffer overflow errors, and cross-site scripting vulnerabilities are among the top three weaknesses.

Table 25: Top 10 CWE software weaknesses by the number of Exploits (D2)

	CWE-ID	Weakness Summary	Num. Exploits
1.	89	SQL Injection	2825
2.	119	Buffer Overflow	2075
3.	79	Cross-site Scripting	1775
4.	94	Code Injection	1033
5.	22	Path Traversal	927
6.	20	Improper Input Validation	703
7.	264	Access Control Error	647
8.	399	Resource Management Error	301
9.	200	Information Disclosure	249
10.	352	Cross-Site Request Forgery	222

4.6 Chapter Summary

In this chapter we have presented a semi-automated method for developing a unified vulnerability knowledge base, i.e., SEVONT: a pyramid of software security vulnerabilities analysis ontologies, which allows us to reconcile and integrate heterogeneous vulnerability data from several SVDBs. Our knowledge modeling approach takes advantage of FCA to guide knowledge engineers in abstracting and reusing concepts across system level ontologies, while at the same time improving knowledge integration and reuse. We illustrated the applicability of our knowledge model through two examples that take advantage of our unified representation to verify the consistency of data in these SVDBs and to enrich existing SVDB knowledge resources by linking them to other resources.

In the next chapter we introduce SV-AF, a framework for integrating SEVONT with other SE ontologies (e.g., source code, build repositories, issue tracker). The objective of SV-AF is to provide a comprehensive, interlinked knowledge base that allows for novel types of cross artifact vulnerability analysis.

Chapter 5

5 SV-AF – A Security Vulnerability Analysis Framework

5.1 Introduction

While software vulnerabilities databases (SVDBs) 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. Several reasons exist for these information silos: (1) A lack of standardized formalism for representing knowledge in the SE 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 chapter introduces a modeling approach that addresses this traceability challenge. More specifically, our approach takes advantage of the SW 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, by using ontologies and Linked Data we can now enrich these repositories with explicit and implicit semantic links and take advantage of SW reasoning services to create true information hubs.

In this chapter, we introduce a Security Vulnerabilities Analysis Framework (SV-AF) which not only establishes traceability links between SVDBs and SE repositories, but also enables practitioners to be notified about potential security vulnerabilities introduced due to the indirect dependencies within their systems. 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 415,604 Maven projects can be identified as potentially affected by these vulnerabilities.

Related work: In a related study, Ilo et al. [63] presented 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[63] package repository. In contrast, our approach has more abstracted and generalizable features which can capture knowledge of different build-systems and package management repositories.

Mircea et al. [164] introduced their Vulnerability Alert Service (VAS) tool to notify users if a vulnerability is reported for a software system. VAS depends on the OWASP Dependency-Check tool which we compare with our SV-AF approach in Section 5.4. 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.

Note: Earlier versions of the work have been published in the 27th IEEE International Symposium on Software Reliability Engineering (ISSRE 2016) [44] and in the journal *Science of Computer Programming*, Volume 121, 2016 [165].

5.2 A Security Vulnerability Analysis Framework

5.2.1 Knowledge Modeling

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

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

ontologies introduced in [20]. More specifically, we extend these ontologies by focusing not only on 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 first model system-specific concepts and iteratively abstracted higher-level shared concepts in upper-ontologies (see Figure 24). The resulting four layer modeling hierarchy is similar to a metadata modeling approach introduced by the Object Management Group (OMG)[64]. 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 [215].

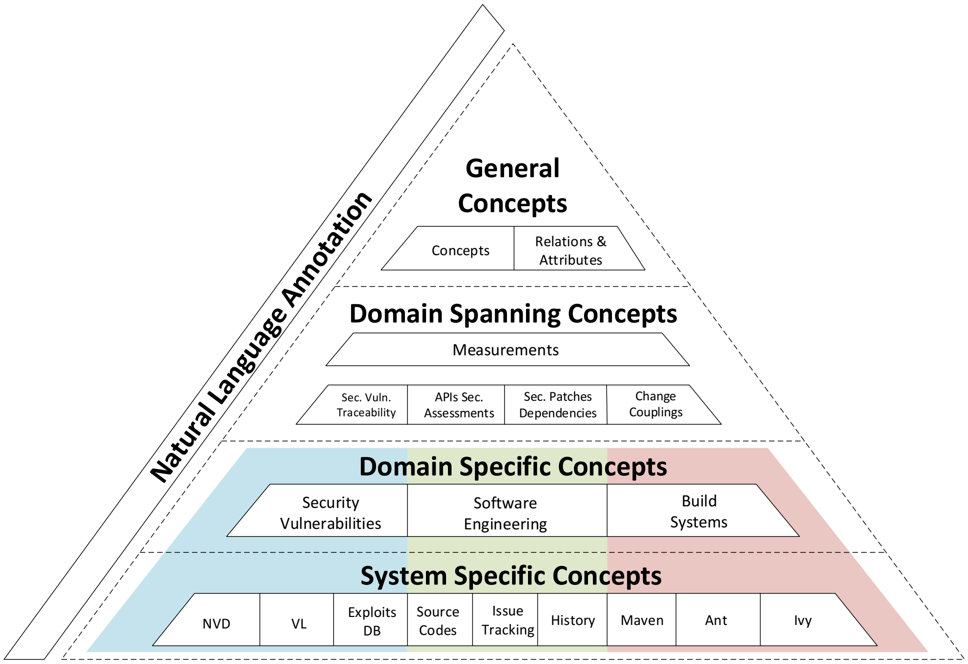


Figure 24: The SV-AF Ontologies Abstraction Hierarchies.

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 of such

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

core concepts are: *Product, Reference, Activity, Stakeholder,* and *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) SEVONT, (2) Software Evolution ONtologies (SEON) [20], 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 system-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.

5.2.2 Knowledge Engineering and Integration

The SW is characterized by decentralization, heterogeneity, and lack of central control or authority. These new features have greatly contributed to the success of the SW, but at the same time they have 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 in our framework (Figure 25).

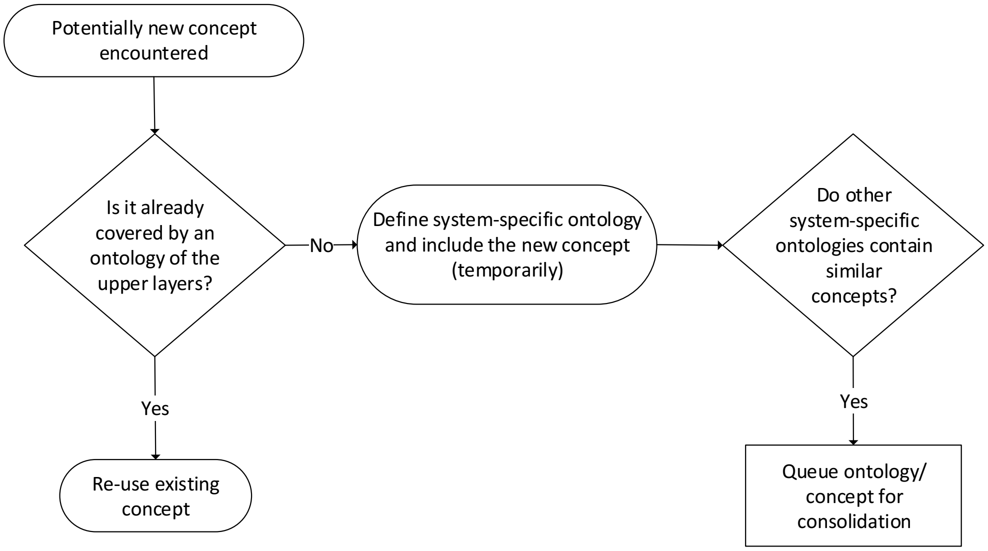

Potentially new concept encountered
Is it already covered by an ontology of the upper layers?
No
Define system-specific ontology and include the new concept (temporarily)
Do other system-specific ontologies contain similar concepts?
Yes
Re-use existing concept
Yes
Queue ontology/ concept for consolidation

Figure 25: 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 been previously modeled in any of the upper SV-AF's layers (e.g., the domain-specific layer) and re-use the existing concepts whenever possible. If no similar concept exists, 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 the knowledge modeling and integration of SE resources, we follow a similar approach as we used for SEVONT. For example, given two similar concepts found in different SE repositories (e.g., issue tracker and build systems), we will first create two distinct system-specific concepts in both ontologies. We then compare these system ontologies and move commonalities to the domain-specific layer, in order to improve knowledge reuse and traceability. Concepts modeled in more than one domain are promoted from the domain-specific to the domain-spanning layer.

5.2.3 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. 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 Figure 26).

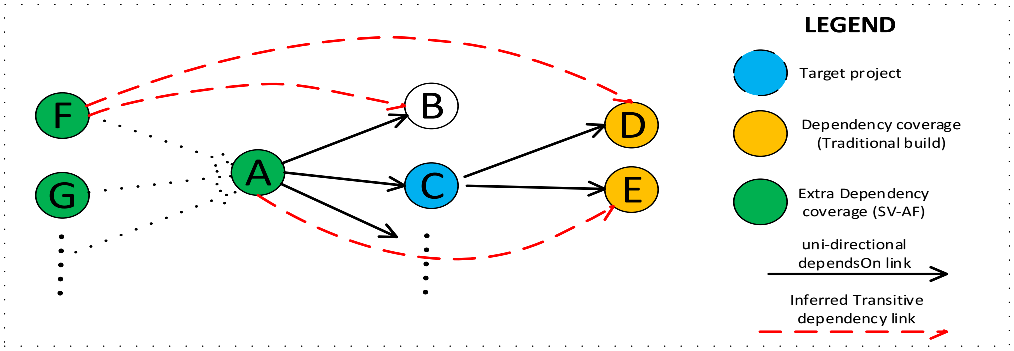

LEGEND
Target project
Dependency coverage (Traditional build)
Extra Dependency coverage (SV-AF)
uni-directional dependsOn link
Inferred Transitive dependency link
F
G
A
B
C
D
E

Figure 26: Unidirectional vs. bi-directional dependencies.

To overcome this challenge, we take advantage of the SW 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 Figure 26, using SBSON we can extend the Maven supported impact analysis on project *C* by not only identifying 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*).

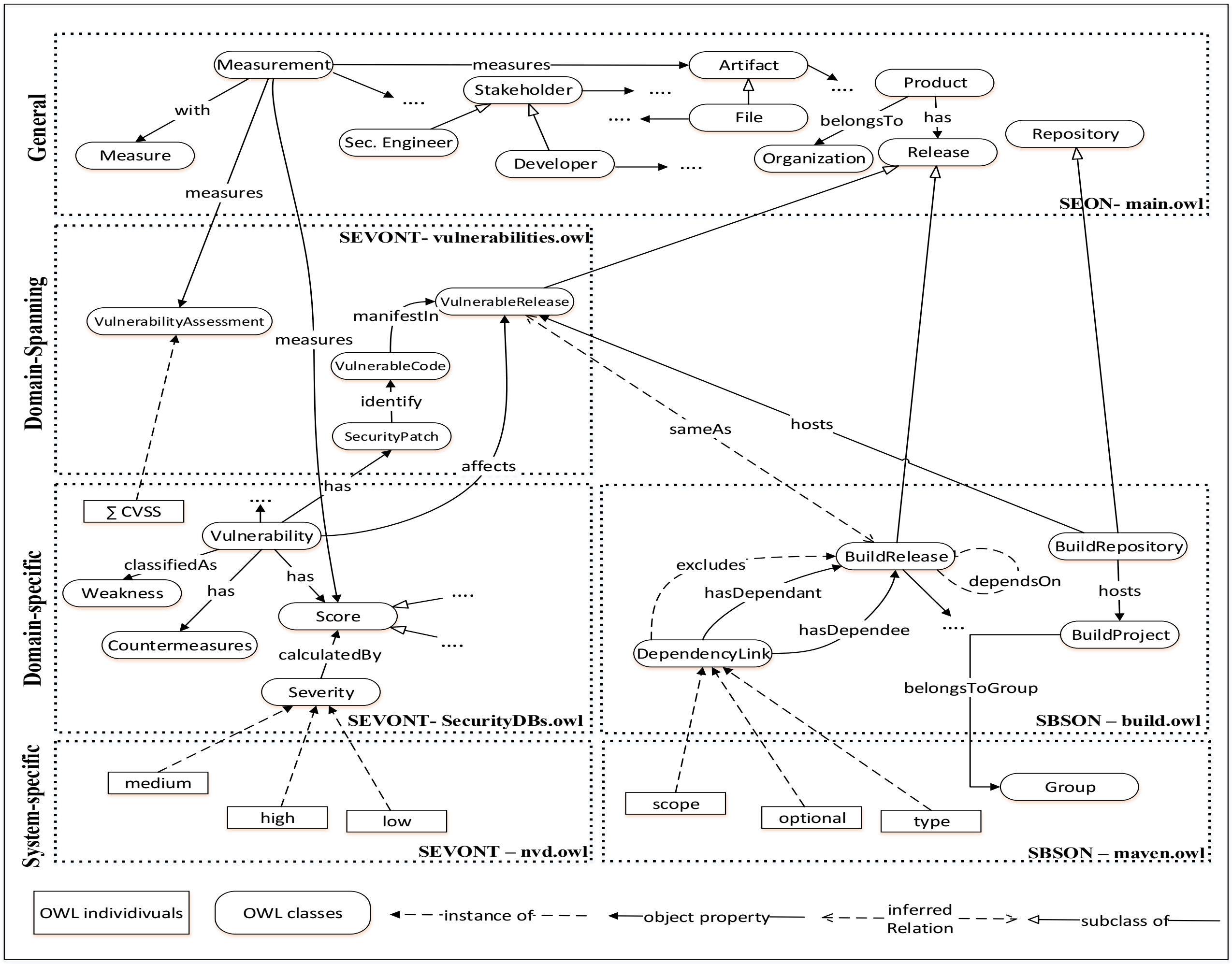


Figure 27: SV-AF's ontologies and concepts involved in software vulnerability dependencies analysis.

As discussed before, our SV-AF knowledge modeling approach allows 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 also provide a concrete usage scenario showing how our unified representation can now support, 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 Figure 27 (data properties have been omitted to improve readability of the figure).

Modeling Vulnerable Release Dependencies: 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*.

5.3 Methodology

5.3.1 Overview

Next, we introduce in more detail our overall methodology (Figure 28) 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.

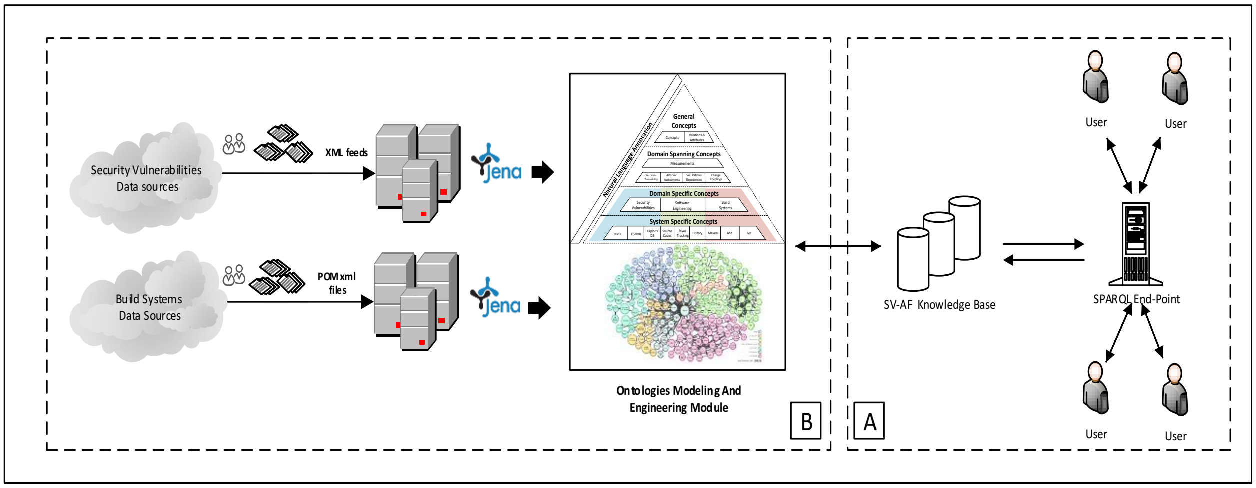


Figure 28: SV-AF system overview.

5.3.2 Fact Extraction and Population

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

The fact extraction process itself consists of extracting facts from the Maven POM files and the NVD XML update feeds (see Figure 28–B). For the ontology population, we use the Jena[65] framework to populate the corresponding artifact ontologies and materialize them using a triple-store.

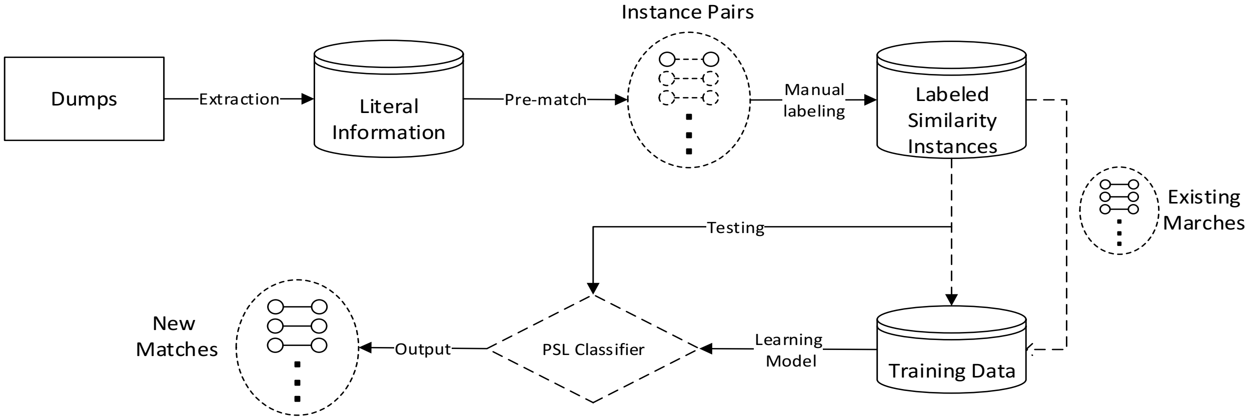


Figure 29: Instances matching approach.

5.3.3 Ontology Instances Alignment

For the alignment of instance in our ontologies, we take advantage of the Probabilistic Soft Logic (PSL) framework [219] which establishes weighted links between ontologies (Figure 29).

PSL uses continuous variables to represent truth values, relaxing the standard Boolean values [219] 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 4).

[65] https://jena.apache.org/

$$\begin{aligned} Rule - 1\text{:}\, type(A, instance) &\wedge type(B, instance) \wedge name(A, X) \\ &\wedge name(B, Y) \wedge similarID(X, Y) \\ &\wedge A.source \neq B.source \\ &\Rightarrow similar(A, B) \text{ wight: } 0.5 \end{aligned}$$

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

Listing 4: PSL rules.

For example, in Listing 4 – Rule 1, the 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 4 – Rule 2, the rule weights are used to indicate that projects with same names and versions are more likely to be similar than projects with same names only (Listing 4 – Rule 1).

Using PSL we can establish owl:sameAs relations between similar instances found in the SEVONT and SBSON ontologies. In this example, two system-specific ontologies, *NVD* and *Maven,* and their corresponding instances $|NVD|$ and $|Maven|$ are used as data sources. 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 purposes, 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 assign these weights to the **owl:sameAs** (see Figure 30) links between the aligned instances and then materialize 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++[66]

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

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

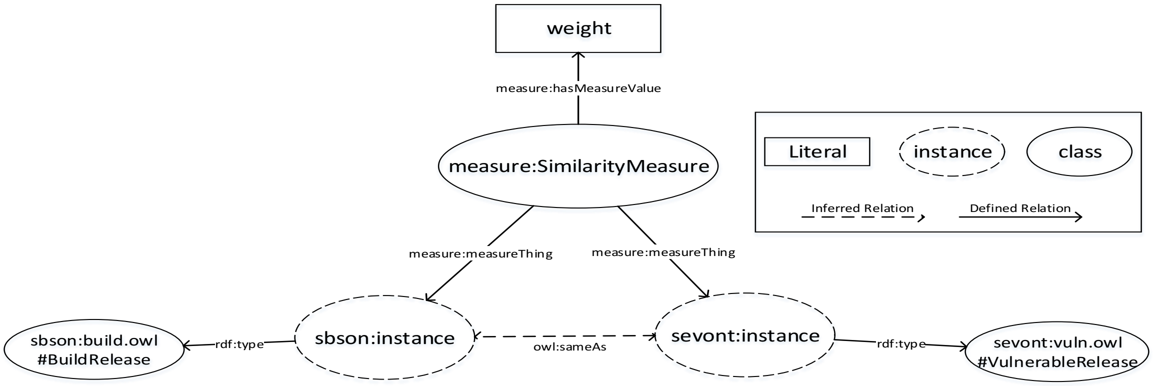


Figure 30: Weighted similarity modeling.

5.3.4 Knowledge Inference 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 Listing 5.

$$\begin{array}{lll}
sevont\!: ProjA & rdf\!: type & sevont\!: VulnerableRelease \\
sbson\!: ProjB & rdf\!: type & sbson\!: BuildRelease \\
sevont\!: ProjA & sevont\!: hasVulnerability & sevont\!: CVE - ID \\
sevont\!: ProjA & \boldsymbol{owl\!: sameAs} & sbson\!: ProjB.
\end{array}$$

Listing 5: owl:sameAs rules example

Given is the following SPARQL query (Listing 6), which takes advantage of the owl:sameAs predicate if inference is enabled

```
REFIX[…]
SELECT ?vulnerability ?release
WHERE{
    ?release rdf:type sbson:BuildRelease.
    ?release sevont:hasVulnerability ?vulnerability
}
```

Listing 6: 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 inferred that ProjB has a vulnerability (CVE-ID[67]) 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 7).

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

Listing 7: SPARQL query returning transitive vulnerable dependencies.

5.4 Case Studies

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

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

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 [42]. 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.

5.4.1 Case Studies Data

For the data collection and extraction in our case studies, we relied on two data sources: the NVD database and the Maven build repository. We downloaded the latest version of the repository from the Maven.org website (Table 26) and downloaded all NVD vulnerability xml feeds from 1990 and 2016 (Table 27). 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 26: Maven Repository statistics

Repository	Projects	Releases	Last Update
Maven [220]	130,895	1,219,731	2016-01-28 16:34:07 UTC

Table 27: Maven Repository statistics

Repository	# unique vulnerabilities	# unique vulnerable products	Period
NVD [221]	74945	109212	1990 - 2016

For case study #2, the objective was to identify the potential transitive impact set of some vulnerable components on other systems. For this study we selected five Apache projects (Table 28) 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 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[68] is a Java implementation of the primary security standards for Web Services, Httpclient[69] is responsible for providing reusable components for client-side authentication, HTTP state management, and HTTP

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

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

connection management. Apache Derby[70] is an open source relational database implemented entirely in Java, Hibernate Validator[71] allows expression and validation of application constraints using annotation-based constraints, and Apache OpenJPA[72] is a Java persistence project that can be used as a stand-alone plain old Java object (POJO) persistence layer or can be integrated into any Java EE compliant container.

Table 28: 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

5.4.2 Case Studies 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 500 randomly selected project instance pairs for which we manually created links. We then executed our PSL alignment rules on this training dataset to train our approach. As a result of 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 similarity functions implemented using a Levenshtein distance metric. In this example, the *SameProject(A,B)* is given a weight of 0.9 (Listing 8), which is based on results from the PSL training set. Figure 31 shows the PSL inference results

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

[71] http://hibernate.org/validator/

[72] http://openjpa.apache.org/

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 8), PSL can now perform *maximum a posteriori (*MPE) reasoning [219] to infer the most likely values for a set of propositions and observed values for the remaining (evidence) propositions.

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

Listing 8: SameProject Rules.

For a full discussion on MPE reasoning, we refer the reader to [219]. The results of the PSL inference are 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.

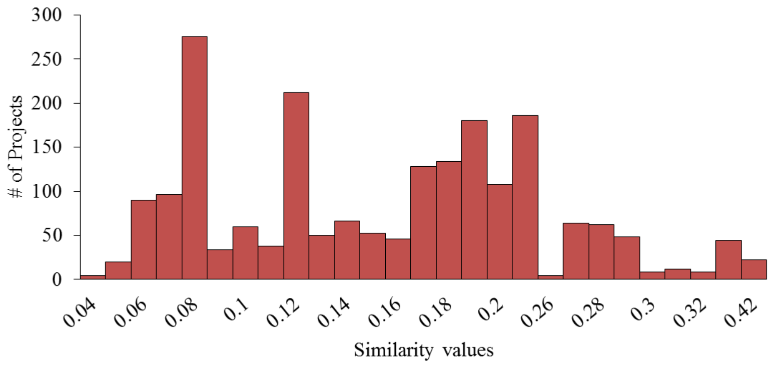


Figure 31: PSL similarities results.

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 5.2.1).

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 29.

Table 29: 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*

A further results analysis showed that projects might often suffer from multiple vulnerabilities. We also observed 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 our dataset, containing 25 known vulnerabilities. Providing this additional insight can guide system update decisions and help avoid the reuse of APIs/components with known security vulnerabilities or components that might be prone to these types of vulnerabilities.

For example, in December 2010, Google released its Nexus S smartphone[73]. The phone was originally running on Android 2.3.3—an Android version that already contained the security vulnerability discussed in Table 30. 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[74] has been reported for five different Android versions (Table 30). For product maintainers this information can help to ensure consistent patching and regression testing across product lines or different versions of a product.

Table 30: 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*

[73] https://en.wikipedia.org/wiki/Nexus_S
[74] https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2013-4787

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 correspond to the number of non-similar instance pairs that are incorrectly classified as same projects. For Recall, false negatives correspond 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.

Table 31: owl:sameAs link (w) evaluation

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

Our analysis (Table 31) 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).

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 [42], a specialized tool which identifies direct dependencies between projects and publicly disclosed vulnerabilities. For the study, we apply the OWASP dependency check tool as our gold standard and compare the detected dependencies against the links generated by our approach (Table 32). 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 32: 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 highlights 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 (Figure 32) that not only identify components that are directly but also indirectly affected by known vulnerabilities. For this impact analysis, we selected 5 open source Java projects (Table 28) with known security vulnerabilities for which we do not distinguish if a component actually makes use (calls) a vulnerable component or not.

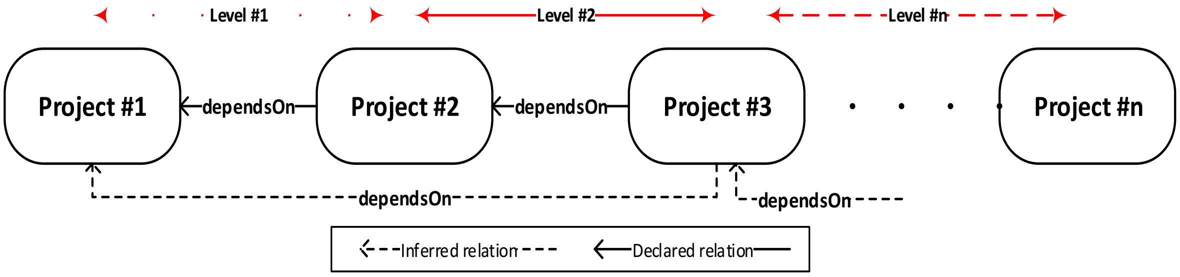


Figure 32: Inferred project dependencies in SBSON.

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 SW. Given the bi-directional links we established between the NVD and the Maven repository, our analysis is no longer limited to identifying whether a project depends on a vulnerable component. Instead, given a vulnerable component we can now also provide a more holistic analysis by identifying in a global context which other projects potentially directly or indirectly depend on this vulnerable component.

Table 33 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 3,805 direct dependent projects (level 1) and 128,109 dependent projects when we consider an additional two levels of transitivity (level 3).

Table 33: 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	4,961	41,326
P3	Derby 10.1.1.0	3	CVE-2005-4849 CVE-2006-7216 CVE-2006-7217	385	37,999	66,147
P4	Hibernate-validator 4.1.0.Final	1	CVE-2014-3558	3,805	39,295	128,109
P5	Openjpa 1.1.0	1	CVE-2013-1768	74	49,460	141,303

Figure 33 illustrates a typical usage scenario for our modeling approach. While the Geronimo-jetty6-javaee5 (version 2.1.1) itself has no known vulnerabilities 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), thus potentially making Geronimo-jetty6-javaee5 a very vulnerable component.

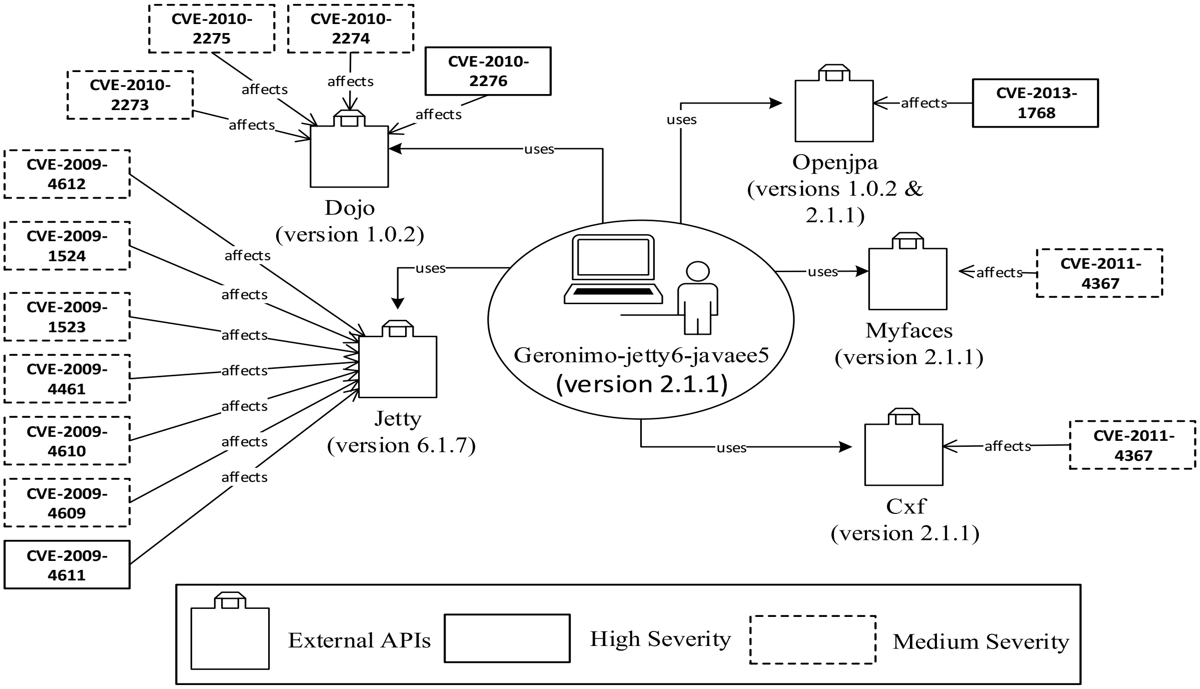


Figure 33: Geronimo-jetty6-javaee5 uses 5 projects (external APIs) from level 1 dependency and each project suffers from security vulnerabilities.

5.5 Discussion and Threats to Validity

As our experiments illustrate, a unified and formal knowledge model can indeed help eliminate 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 humans or machines. In addition, rather than relying on proprietary analysis solutions, our modeling approach takes advantage of the SW technology stack, including standardized knowledge representation and inference services.

5.5.1 Case Study 1

Identifying known security vulnerabilities in software projects has been widely discussed in the literature [43], [164], [165], [222]. 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 is not explicitly found in any of these resources; and (3) given the bi-directional links, our analysis can go beyond traditional inter-project dependencies and include intra-project dependencies. The results from our case study show that the problem of depending on vulnerable third party components with known security vulnerabilities is a common and widespread problem [165].

5.5.2 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 the sharing and integration 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 the transformation of 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 7 will not only return all projects that directly depend on a vulnerable component, but also those that are indirectly dependent on that component.

5.5.3 Threats to Validity

A threat to the internal validity of the study is that the instance pair matches for our training set were manually created and thus potentially prone to human error. 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 34 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 little adverse effect on our results.

Table 34: Dataset size evaluation

Data Points	**SV-AF (w=0.1)**			
	Precision	**|Distance from Average|**	**Recall**	**|Distance from Average |**
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**	-

Other threats to validity related to the ontology's design quality are discussed in detail in Chapter 8.

5.6 Chapter Summary

In this chapter, we introduced a Security Vulnerabilities Analysis Framework (SV-AF) 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 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. We also provided a discussion on how our framework differs from related work in the domain.

In the next chapter, we discuss how SV-AV supports the recovery of traceability links between APIs and vulnerability information.

Chapter 6

6 Recovering Semantic Traceability Links between APIs and Security Vulnerabilities

6.1 Introduction

Vulnerabilities found in APIs no longer affect only individual projects but instead might spread across projects and even the global software ecosystem. Tracing such vulnerabilities at a global scale becomes an inherently difficult task since many of the resources required for such analysis still rely on proprietary knowledge representation. A report [43] shows that 88% of the code in today's applications comes from OSS libraries and frameworks; with 26% of these OSS frameworks/libraries having known vulnerabilities which often remain undiscovered. In 2017, "Using Components with Known Vulnerabilities" [223] was ranked 9th in the OWASP Top Ten [224] list of software security flaws.

Current approaches for ensuring secure software fall into two main categories. The first category requires organizations to create barriers that prevent developers and end-users from performing potentially risky actions, e.g., runtime protection. While this approach can reduce 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 SW 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 current analysis approaches must contend with and introduce new types of vulnerability analysis at a global scale.

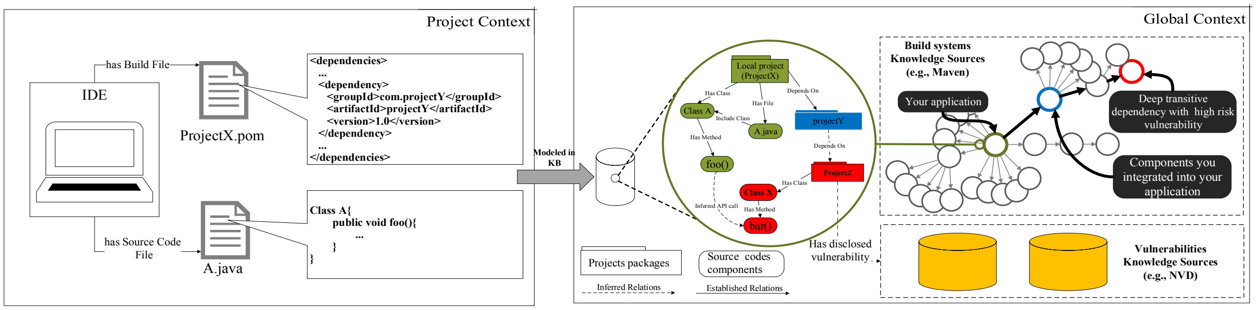
In Chapter 5 we introduced SV-AF, our modeling approach to establish traceability links between SE repositories and SVDBs. In this chapter, we extend our previous SV-AF with knowledge from version control systems (VCS) repositories to provide additional analysis services such as: (1) identifying and 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.

Related work: Several approaches for static vulnerability analysis and vulnerability detection in source code exist (e.g., [156], [133]). Plate et. al [156] proposed a technique that supports the impact analysis of vulnerability based on code changes introduced by security fixes. Their approach relies on dynamic analysis to determine if a vulnerable code was executed within a given project. While our approach relies on static analysis and might be less precise, it delivers a more holistic approach which not only considers all possible executions but also supports a more general intra- and inter-project dependency analysis. Furthermore, our approach allows us to take advantage of semantic reasoning services to infer implicit facts about vulnerable code usages and supports bi-directional dependency analysis —including impacts to external dependencies.

Nguyen et. al [133] 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.

6.1.1 Motivating Example

Existing research on recommending APIs to developers (e.g., [225]) has focused on recommending potentially useful APIs to developers to reduce development and testing time.



```
<dependencies>
  ...
  <dependency>
    <groupId>com.projectY</groupId>
    <artifactId>projectY</artifactId>
    <version>1.0</version>
  </dependency>
  ...
</dependencies>
```

```
Class A{
      public void foo(){
            ...
      }
}
```

Figure 34: Integrating code and build information with knowledge from other heterogeneous resources.

For example, in [225] the authors explicitly recommend developers 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 authors in [225] actually recommended a version of Apache Derby which has two known security vulnerabilities (Table 35). These known vulnerabilities had already been published in the National Vulnerability Database (NVD) repository.

Table 35: 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 authors of the paper were 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., SW and its technology stack), novel analysis approaches across knowledge boundaries at both the intra- and inter-project level can be introduced.

For example, Figure 34 shows an example of an IDE with an open Maven[75] 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. 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 SW, we can now infer direct and indirect dependencies for the local project (ProjectX in Figure 34). In addition, giving the bi-directional links in our modeling approach, we can expand our analysis to a global scale to answer questions like: 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 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 can be extended to not only recommend suitable APIs but to now recommend suitable APIs with **no known** direct/indirect vulnerabilities or to automatically notify developers when an already used API **becomes exposed to a potential vulnerability**.

Note: An earlier version of the work done in this chapter was published in the 10th IEEE International Conference on Software Testing, Verification and Validation (ICST 2017) [166].

6.2 Modeling API Vulnerabilities

It is generally accepted that inadvertent programming mistakes can lead to software security vulnerabilities and attacks [43]. Mitigating these vulnerabilities can become a major challenge

[75] http://search.maven.org/

for developers since not only their own source code might contain exploitable code, but so might 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 levels (Figure 35). Our methodology consists of three major steps: knowledge modeling, alignment of ontologies, and knowledge inferences and reasoning.

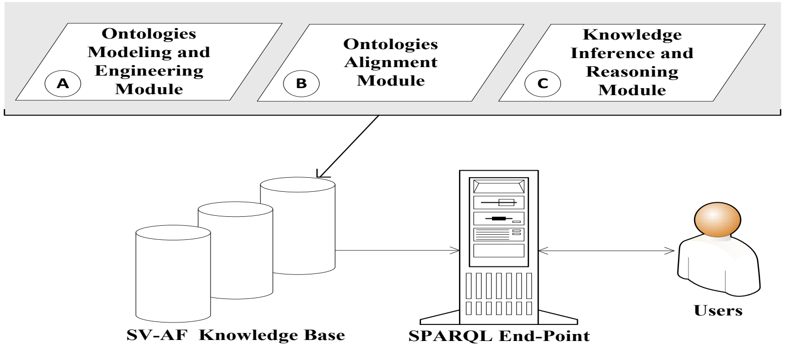

A
Ontologies Modeling and Engineering Module
B
Ontologies Alignment Module
C
Knowledge Inference and Reasoning Module
SV-AF Knowledge Base
SPARQL End-Point
Users

Figure 35: System overview.

6.2.1 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 Chapters 4 and 5. In our modeling approach 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 including semantic relations and properties that allow us to take advantage of inference services provided by the SW.

For our model we followed a bottom-up modeling approach, where we first extracted system specific concepts and then iteratively abstracted shared concepts in upper ontologies (see Chapter 5).

To improve the readability of the chapter, 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 [215]. Figure 36 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.

6.2.2 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.

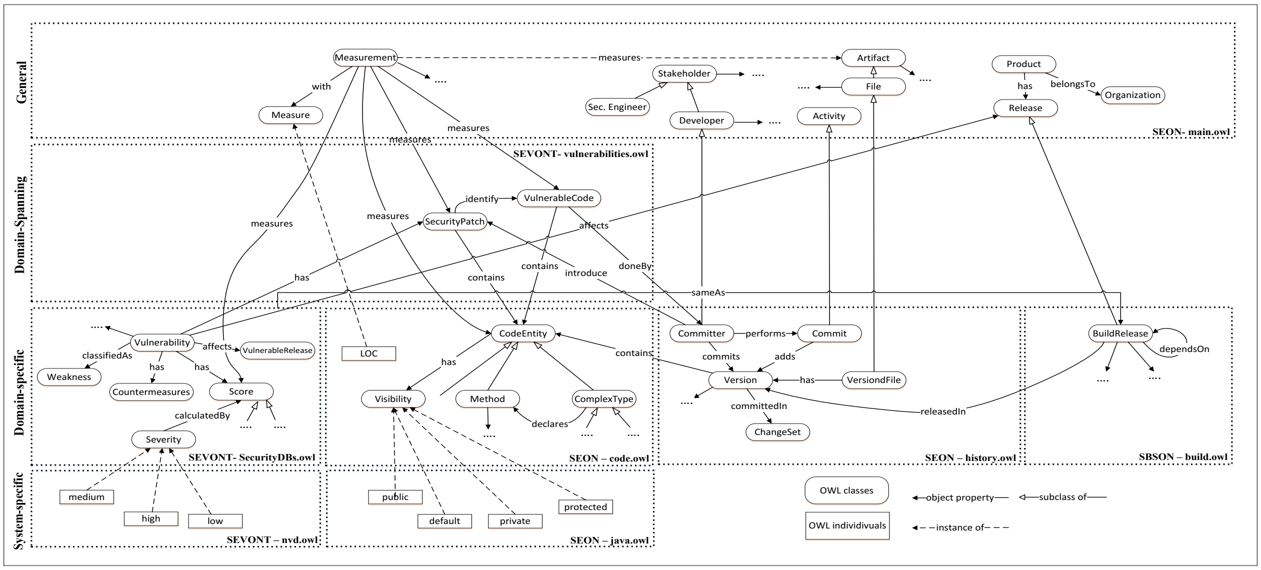


Figure 36: The SV-AF's ontologies concepts involved in an API.

Alignment of SEVONT and SBSON Ontologies. In uncertain graphs [226], edges are associated with uncertainties; edges measure 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 [219]. For example, in Figure 37, the project instance $V_m$from SBSON graph is similar to vulnerable product instance V_n from SEVONT graph through $owl{:}sameAs$ (*ω(e)*) edge.

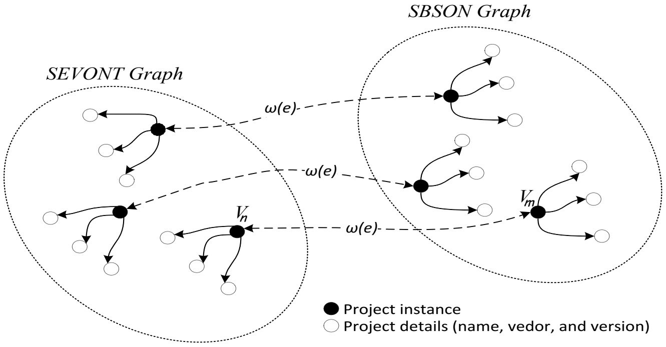


Figure 37: SV-AF knowledge base similarity graphs.

Note that m and n represent the projects original data sources, Maven and NVD respectively. Additional explanations of how the $owl{:}sameAs$ weights are created and how PSL is implemented and tested to establish the sematic links are discussed in Chapter 5.

Alignment of SEVONT and SEON Ontologies. For this alignment, we extend the process discussed in Chapter 5 to also include 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 SW to infer implicit relationships between vulnerabilities and commits. More specifically, for the alignment we take advantage of Semantic

Web Rule Language (SWRL)[76] rules (Listing 9) to establish links between vulnerability and commit instances. This alignment will take place if any of the following 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 9: 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 [44].

6.2.3 Knowledge Inferencing and Reasoning

The SW stack includes a scalable, persistent knowledge storage infrastructure. Triple-stores[77] not only provide data persistence but also support some basic scalable inference on big data (e.g., RDFS, RDFS++) [44]. 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 35) from our illustrative queries and rules to improve their readability.

Table 36: 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#

[76] https://www.w3.org/Submission/SWRL/
[77] 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 [13].

Same-As inference: As we discussed in Chapter 5 (section 5.3.4), 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 10), we can now take advantage of the $owl{:}\,sameAs$ predicate (if inference is enabled):

```
PREFIX[…]
SELECT ?vulnerability ?release
WHERE{
    ?release rdf:type sbson:BuildRelease.
    ?release sevont:hasVulnerability ?vulnerability
}
```

Listing 10: 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 [227], [228]. 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 11).

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

Listing 11: 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 [229]. For example, a *Method* or *Class* is a subconcept of a *CodeEntity*. Subsumption hierarchies add significant power to ontologies in global source code (APIs) analysis [25] 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 [229]: 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. Figure 38 shows the property hierarchy we use to model source code dependencies. Given this property hierarchy and the subsumption inference, a simple query (Listing 12) 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*).

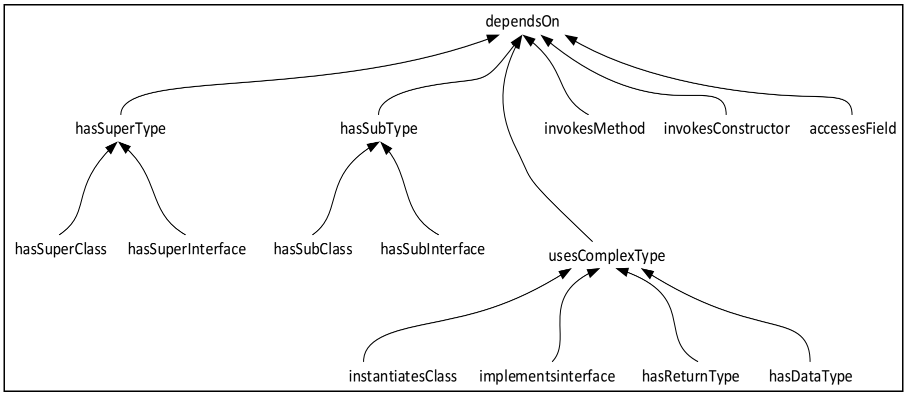


Figure 38: Hierarchy of code properties.

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

Listing 12: dependsOn subsumption query.

6.3 Case Study

In the next sections, we discuss the applicability of our modeling approach in tracing and analyzing known vulnerabilities at intra- and inter-project levels.

6.3.1 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 SW 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 (Figure 39).

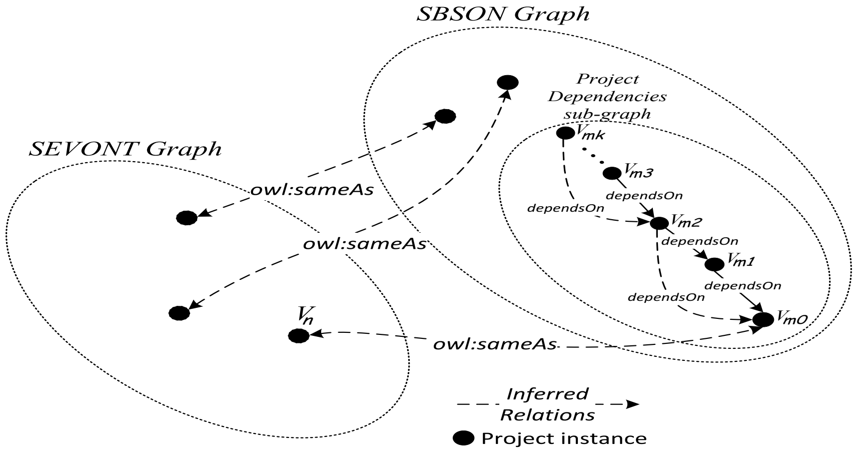


Figure 39: Inferred project dependencies in SBSON.

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[78]: "*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'*".

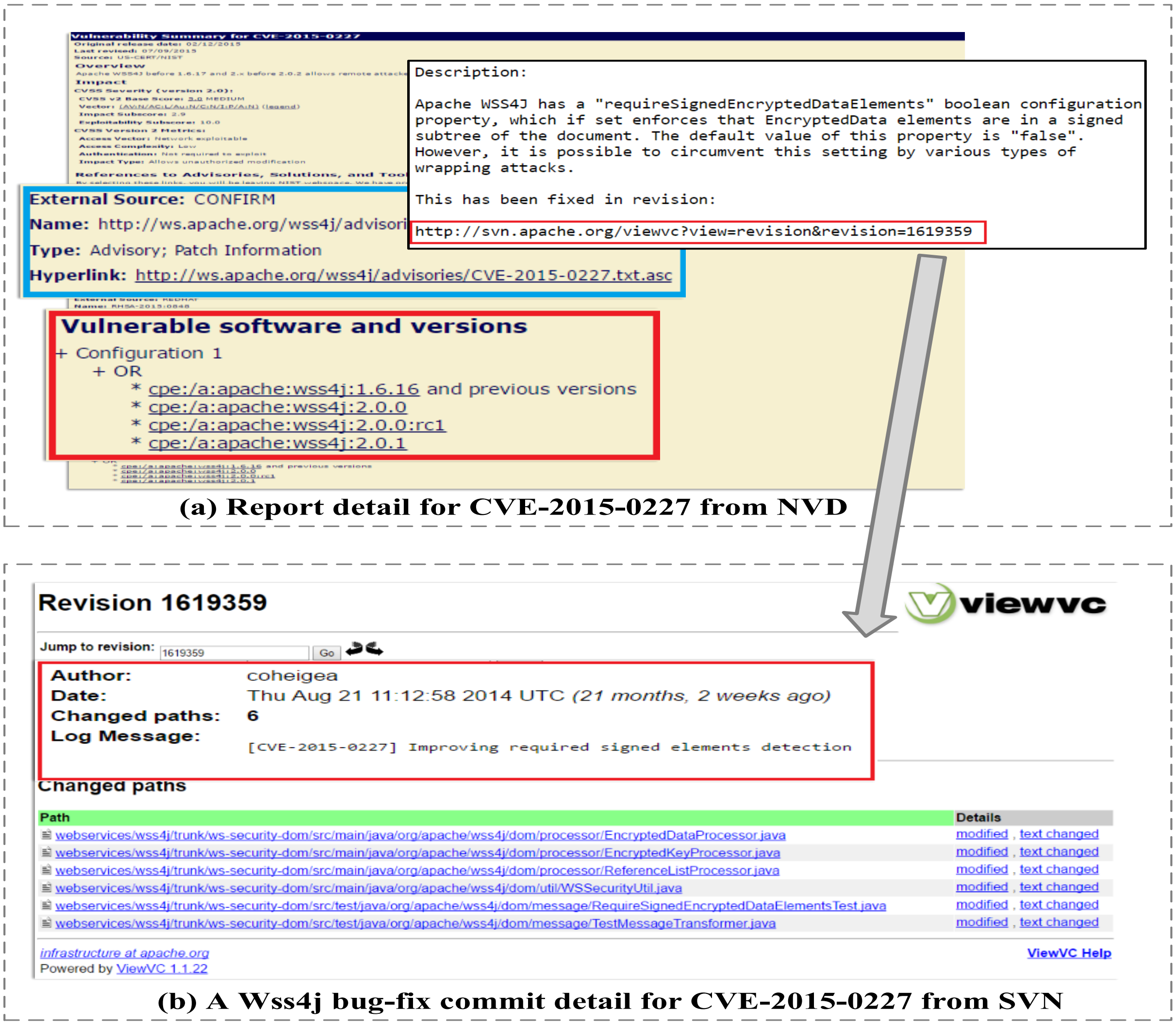
This vulnerability affects the management of permissions, privileges, and other security features that are used to perform access control to Apache WSS4J versions before 1.6.17 and to version 2.x before 2.0.2.

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 must now 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 to an entry in a versioning repository. We then extract all commit logs within the versioning repository that have a reference to a CVE-ID. Figure 40 shows an example of such a commit log message entry: " *[CVE-2015-0227] Improving required signed elements detection.*"

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



Vulnerability Summary for CVE-2015-0227
Original release date: 02/12/2015
Last revised: 07/09/2015
Source: US-CERT/NIST
Overview
Apache WSS4J before 1.6.17 and 2.x before 2.0.2 allows remote attacke
Impact
CVSS Severity (version 2.0):
CVSS v2 Base Score: 5.0 MEDIUM
Vector: (AV:N/AC:L/Au:N/C:N/I:P/A:N) (legend)
Impact Subscore: 2.9
Exploitability Subscore: 10.0
CVSS Version 2 Metrics:
Access Vector: Network exploitable
Access Complexity: Low
Authentication: Not required to exploit
Impact Type: Allows unauthorized modification
References to Advisories, Solutions, and Tool

```
Description:

Apache WSS4J has a "requireSignedEncryptedDataElements" boolean configuration
property, which if set enforces that EncryptedData elements are in a signed
subtree of the document. The default value of this property is "false".
However, it is possible to circumvent this setting by various types of
wrapping attacks.

This has been fixed in revision:

http://svn.apache.org/viewvc?view=revision&revision=1619359
```

External Source: CONFIRM
Name: http://ws.apache.org/wss4j/advisori
Type: Advisory; Patch Information
Hyperlink: http://ws.apache.org/wss4j/advisories/CVE-2015-0227.txt.asc

Vulnerable software and versions
+ Configuration 1
  + OR
    * cpe:/a:apache:wss4j:1.6.16 and previous versions
    * cpe:/a:apache:wss4j:2.0.0
    * cpe:/a:apache:wss4j:2.0.0:rc1
    * cpe:/a:apache:wss4j:2.0.1

(a) Report detail for CVE-2015-0227 from NVD

Revision 1619359

Jump to revision: 1619359 Go

Author: coheigea
Date: Thu Aug 21 11:12:58 2014 UTC *(21 months, 2 weeks ago)*
Changed paths: 6
Log Message:
`[CVE-2015-0227] Improving required signed elements detection`

Changed paths

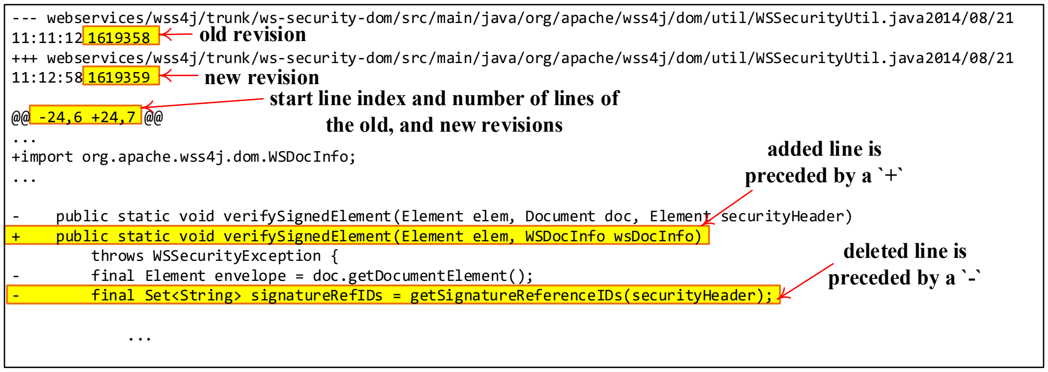
Path	Details
webservices/wss4j/trunk/ws-security-dom/src/main/java/org/apache/wss4j/dom/processor/EncryptedDataProcessor.java	modified , text changed
webservices/wss4j/trunk/ws-security-dom/src/main/java/org/apache/wss4j/dom/processor/EncryptedKeyProcessor.java	modified , text changed
webservices/wss4j/trunk/ws-security-dom/src/main/java/org/apache/wss4j/dom/processor/ReferenceListProcessor.java	modified , text changed
webservices/wss4j/trunk/ws-security-dom/src/main/java/org/apache/wss4j/dom/util/WSSecurityUtil.java	modified , text changed
webservices/wss4j/trunk/ws-security-dom/src/test/java/org/apache/wss4j/dom/message/RequireSignedEncryptedDataElementsTest.java	modified , text changed
webservices/wss4j/trunk/ws-security-dom/src/test/java/org/apache/wss4j/dom/message/TestMessageTransformer.java	modified , text changed

infrastructure at apache.org
Powered by ViewVC 1.1.22 **ViewVC Help**

(b) A Wss4j bug-fix commit detail for CVE-2015-0227 from SVN

Figure 40: 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 [133]. In our approach, we use the standard diff command to identify the vulnerable code fragments by comparing it with its unpatched version. Figure 41 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 Figure 36).



```
--- webservices/wss4j/trunk/ws-security-dom/src/main/java/org/apache/wss4j/dom/util/WSSecurityUtil.java2014/08/21
11:11:12 1619358
+++ webservices/wss4j/trunk/ws-security-dom/src/main/java/org/apache/wss4j/dom/util/WSSecurityUtil.java2014/08/21
11:12:58 1619359
@@ -24,6 +24,7 @@
...
+import org.apache.wss4j.dom.WSDocInfo;
...

-    public static void verifySignedElement(Element elem, Document doc, Element securityHeader)
+    public static void verifySignedElement(Element elem, WSDocInfo wsDocInfo)
         throws WSSecurityException {
-        final Element envelope = doc.getDocumentElement();
-        final Set<String> signatureRefIDs = getSignatureReferenceIDs(securityHeader);

             ...
```

Figure 41: Diff output for WSS4J r1619358 and r1619359.

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 9) to the commit containing the patch.

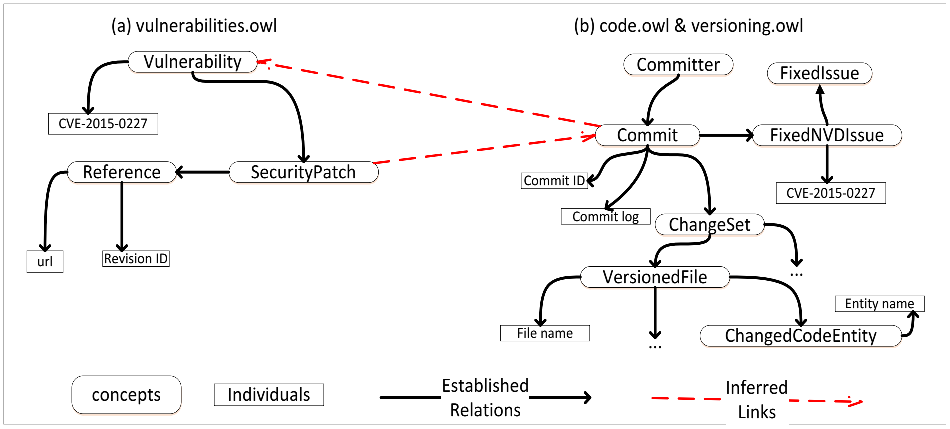


Figure 42: Inferred links between vulnerabilites.owl, code.owl, and versioning.owl.

Based on the inferred links (see Figure 42) and using the SPARQL query in Listing 13, 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.

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

Listing 13: Query to retrieve vulnerable code fragments across project boundaries.

Findings: Table 36 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 12 and 13). Table 37 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 37: 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. Figure 43 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-populated the source code ontologies with the new (modified) code facts and again invoke the same query we used earlier in the case study.

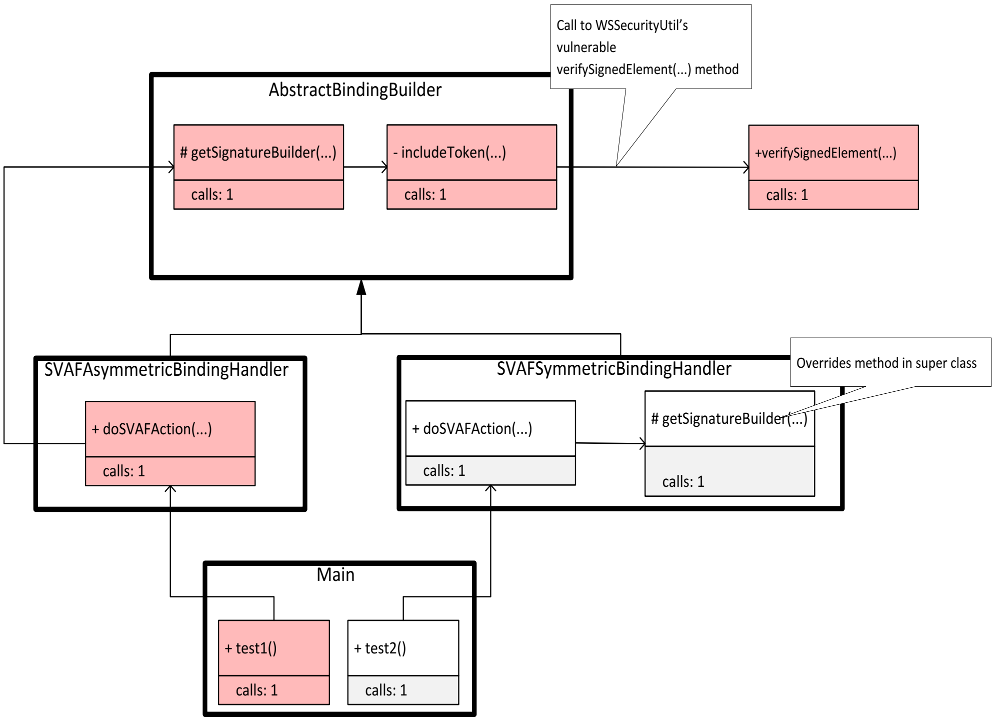

Call to WSSecurityUtil's vulnerable verifySignedElement(...) method
AbstractBindingBuilder
# getSignatureBuilder(...)
calls: 1
- includeToken(...)
calls: 1
+verifySignedElement(...)
calls: 1
SVAFAsymmetricBindingHandler
+ doSVAFAction(...)
calls: 1
SVAFSymmetricBindingHandler
Overrides method in super class
+ doSVAFAction(...)
calls: 1
# getSignatureBuilder(...)
calls: 1
Main
+ test1()
calls: 1
+ test2()
calls: 1

Figure 43: Class diagram for our modified package.

The results of this query are shown in Table 38 which includes the classes within our modified project that directly or indirectly invoke the vulnerable method `WSSecurityUtil.verifySignedElement`.

Table 38: 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 the 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.

6.3.2 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) [42], which is an open source tool, and a closed-source tool from SAP labs [156].

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 [156].

Given that the OWASP DC tool does distinguish whether a vulnerable library code is used or not, we limit our comparison to: “ide*ntify if a project depends on libraries with disclosed vulnerabilities independent of the use of the vulnerable source code”*. Table 39 reports the results from our comparison, which include 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 39: 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

6.4 Findings

As our case study illustrates, our ontology-based knowledge modeling approach can integrate information originating from different heterogeneous knowledge resources. Next, we discuss how our approach overcomes a number of challenges identified with the 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, often making the linking of entities across knowledge resources 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 affect 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 precision (e.g., 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 13) 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 [230], the risk of such transitive vulnerable method invocations increases.

Information silos challenges. Although both analysis tools SAP and OWASP DC link 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 of unique resource identifiers (URIs), facilitating integration with other knowledge resources, even at a global scale.

6.5 Chapter Summary

This chapter 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 SE domain. This formal knowledge representation allows us to take advantage of inference services provided by the SW, providing additional flexibility compared to traditional proprietary analysis approaches.

In the next chapter we will present another application of using SV-AF; it is an extension contribution of this chapter, in which we propose a semantic trustworthiness model for measuring APIs security impacts.

Chapter 7

7 API Trustworthiness: An Ontological Approach For Software Library Adoption

This chapter introduces another application of our SEVONT model, this time focusing on assessing the trustworthiness of software systems. This Ontology-Based Trustworthiness Assessment Model (OntTAM) is a continuation of our previous SE-EQUAM assessment model [231]. For the context of this research we extend the original SE-EQUAM model for the domain of software library trustworthiness. More specifically, we show how the SV-AF can be extended to integrate with vulnerability trustwothiness measurement to assess libraries in terms of their API breaking changes, security vulnerabilities, license violations, and their potential impact on client projects.

7.1 Introduction

Traditional software development processes, with their focus on closed architectures, platform-dependent tools, and software, restrict potential code reuse. With the introduction of the Internet these restrictions have been removed, allowing for global access, online collaboration, information sharing, and internationalization of the software industry [232]. Software development and maintenance tasks can now be shared amongst team members working across and outside organization boundaries. Code reuse through resources such as software libraries, components, services, design patterns, and frameworks published on the Internet have become an essential aspect of this global code reuse and sharing among developers and organizations within the SE industry. Most of today's software projects increasingly depend on the use of external libraries which allow software developers to take advantage of features provided by Application Programming Interfaces (APIs) without having to reinvent the wheel. Unfortunately, even though third-party libraries are readily available, developers are faced with new challenges with this new form of code reuse, such as being unaware of the existence of libraries, selecting the most relevant library among several possible alternatives, and how to use features provided by these libraries [233], [234].

Several software library recommendation approaches have been proposed to address these challenges. These approaches fall into two main categories: (1) recommendation systems for libraries and APIs based on characteristics such as popularity [230], frequency of migration [235], [236], and stability [237], without considering the context of use of these libraries; and (2) techniques that take a client's context into account when recommending libraries (e.g., using the history of method usages by developers [238]).

However, reused software libraries should not only satisfy a client's functional requirements; they must also satisfy non-functional requirements (NFR) such as security, safety, and dependability [239], which are critical to the success of software systems. NFRs are often referred to as system qualities and can be divided into two main categories: (1) execution qualities—qualities which are observable at run time (e.g., performance and usability); and (2) evolution qualities, such as testability, trustworthiness, maintainability, extensibility, and scalability, which are embodied in the static structure of a software system. NFRs often play a critical role in the acceptance and trust users will have in a final software product. However, assessing and evaluating trustworthiness of today's software systems and software ecosystems remains a challenge due to issues ranging from a lack of traceability among software artifacts to limited tool support.

Trustworthiness is also an inherently subjective and ubiquitous term since its interpretation depends on the assessment context of the stakeholder, which might be different among stakeholders, and the context of use in which the library is used. Assessment models, therefore, should provide the flexibility and customizability to take into account such specific application contexts and the particular assessment needs of stakeholders [231].

The work in this chapter is a continuation of our previous work on semantic modeling and tracing of software security vulnerabilities (Chapter 5), semantic analysis (Chapter 6), and quality assessment (SE-EQUAM) [231]. In what follows, we present our Ontology-Based Trustworthiness Assessment Model (OntTAM) which is an instantiation and extension of our SE-EQUAM assessment model [10] for the domain of software library trustworthiness.

More specifically, we illustrate how OntTAM can be instantiated to take advantage of our existing unified knowledge representation of different SE related knowledge resources and support an automated analysis and assessment of trustworthiness quality attributes of libraries. We argue that ontologies not only promote and support the conceptual representation of knowledge resources in software ecosystems, but also let us take advantage of semantic reasoning during the assessment of trustworthiness quality factors. Furthermore, our modeling approach allows for the customization of the trustworthiness assessment model to reflect specific assessment needs while at the same time facilitates the comparison of trustworthiness across projects by defining a standard set of measures and sub-factors.

Our research is significant for several reasons:

1) We introduce OntTAM, a novel trustworthiness assessment model that takes advantage of both our previous generic SE-EQUAM software assessment model [231] and our unified ontological knowledge representation of different SE related knowledge resources (discussed in Chapters 5, 6, and [231]), while supporting the customization of the model to meet a stakeholder’s assessment needs.
2) We introduce, as part of OntTAM, novel trustworthiness measures which measure API breaking changes, security vulnerabilities, and license violations. These measures take advantage of our ontologies and semantic reasoning services to allow for a trustworthiness analysis across the boundaries of individual artifacts and projects.
3) We report on a case study that illustrates how our approach can be applied to assess the trustworthiness of OSS libraries, and discuss the potential impact of these libraries on the trustworthiness of the overall system.

7.1.1 Motivation Example

Here, we introduce a motivating example (Figure 44) which is an extension of our example used in Chapter 6, describing how our fictional software developer (Bob) attempts to re-use external libraries while facing several challenges in selecting the best library for his project and trying to reduce their negative effect on the trustworthiness of his own project.

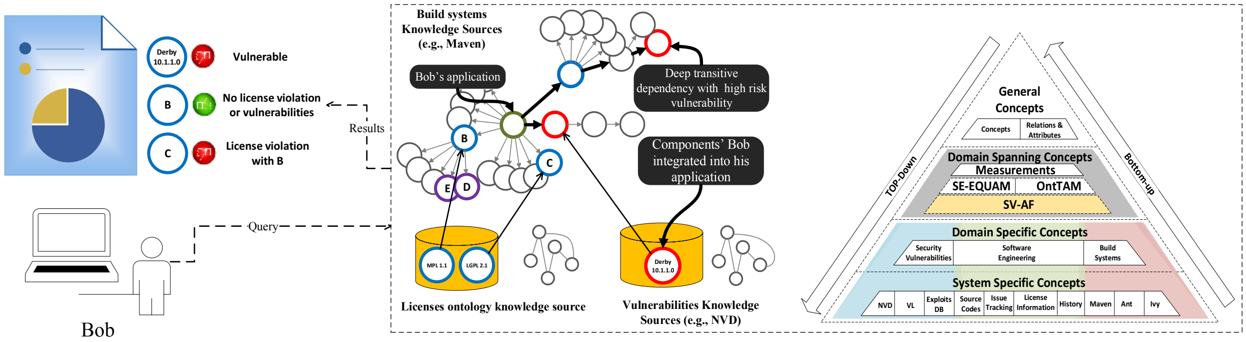


Figure 44: Motivating Example – How OntTAM can assist developers in trust assessment.

Bob is currently developing an application which requires an embedded database. Bob tries to reduce his development effort by searching the Internet for possible third-party libraries and components which meet his work context. His search leads return Apache Derby[79], an open source embedded DBMS implemented entirely in Java. However, Bob is now faced with the dilemma of deciding upon which version of Derby he should use—the most recent (Derby version 10.11.1.1) or the most widely used one (Derby version 10.1.1.0). Following the recommendations published in existing research (e.g., Mileva et al. [225]), Bob decides to use an older version of Apache Derby (version 10.1.1.0) due to its widespread usage/popularity. However, this recommendation results in the reuse of a component which contains three known security vulnerabilities that are already reported in the National Vulnerability Database (NVD) (Table 40). In contrast, the newer version of Derby (version 10.11.1.1) would not have contained any known vulnerabilities.

[79] db.apache.org/derby/

Table 40: Example of Derby versions and their depedent projects in Maven

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

However, this is not the only risk Bob is susceptible to when selecting a library. Derby is licensed under the Apache 2 copyright license; for Bob not to introduce any license violation or incompatibility, he has to make sure that the selected library is compliant with his project license. For example, one cannot combine code released under the Apache 2 license with code released under the GNU GPL 2 [240].

As this example illustrates, several quality-related issues with the reuse of a third-party library can arise and they are often difficult to discover by the user since the relevant information is spread across multiple knowledge resources. The problem is further exaggerated by the large number of additional transitive dependencies which are introduced by these third-party libraries and their dependencies. A vulnerability or license violation might not only occur directly between Bob's project and the Derby library, but also between Bob's project and one of the libraries the Derby library depends on.

7.2 Background

7.2.1 External Library Re-Use and its Implications on Project Quality

As previously discussed, reuse of functionality provided by third-party (external) software libraries is a growing trend in the software development industry. Automated dependency management features have been introduced in modern build systems to simplify the integration and reuse of external libraries during development. Developers no longer have to manually manage their dependencies on software libraries. Build systems and dependency management tools automatically download and manage all required dependent components (including transitive dependencies) and perform any necessary dependency mediation (conflict resolution) when multiple versions of a dependency are encountered. Although this relieves developers of some dependency management, there remains an increasing risk of including libraries which can negatively affect a project's overall quality and trustworthiness. In our research, we particularly consider the following quality risks introduced by software libraries: API breaking changes, security vulnerabilities, and license violations. In what follows, we briefly introduce background information about API breaking changes and license violations.

7.2.1.1 API Breaking Changes

Software libraries take advantage of visibility modifiers (e.g., public and protected modifiers in Java) to provide reusable and extendable APIs to other applications. However, these software libraries, as other software components, are subject to change as they evolve over time. Unfortunately, the cost of evolving libraries may become higher, since such changes might impact many external clients. API changes can be classified into breaking and non-breaking changes (see Table 41) and can be defined as follows [241]:

- *Breaking changes*. Any change that breaks backward-compatibility through removal or modification of API elements, resulting in compilation errors in client projects after the API update.
- *Non-breaking changes*. Changes that preserve compatibility and usually involve the addition of new functionalities to the library. Thus, allowing migration between API versions which include only non-breaking changes does not cause negative effects to client applications.

Table 42 shows the top 10 breaking and non-breaking changes in the Maven repository as reported by [242]. These breaking changes are obtained from 126,070 pairs of current and next versions of software libraries hosted in the Maven repository. The most frequently observed breaking changes are method removals (177,480 observed occurrences). A method removal is considered to be a breaking change if the removal leads to compilation errors in places where this method is used. The most common non-breaking API changes are method additions, with 518,690 occurrences. Although performing a change to a library might be a straightforward task, resulting breaking changes can have a significant ripple effect which often will not only affect a single dependent class but even the complete ecosystem.

Table 41: Breaking and non-breaking changes

Category	API Element	List of Changes
Breaking	Type	Removal, Visibility Loss, Super-type change
	Field	Removal, Visibility Loss (e.g., public to private), Type change (e.g., double to integer), Default value change
	Method	Removal, Visibility Loss, Return type change (e.g., boolean to void), Parameter list change, Exception list change
Non-breaking	All	Addition, Visibility gain (e.g., from private to public or protected), Deprecation (e.g., deprecated method removal)

Table 42: The most common breaking and non-breaking changes in the Maven Repository [242]

Breaking changes			Non-breaking changes		
#	Change type	Frequency	#	Change type	Frequency
1	Method removed	177,480	1	Method added	518,690
2	Class removed	168,743	2	Class added	216,117
3	Field removed	126,334	3	Field added	206,851
4	Parameter type change	69,335	4	Interface added	32,569
5	Method return type change	54,742	5	Method removed, inherited still exists	25,170
6	Interface removed	46,852	6	Field accessibility increased	24,954
7	Number of arguments changed	42,286	7	Value of compile-time constant changed	16,768
8	Method added to interface	28,833	8	Method accessibility increased	14,630
9	Field type changed	27,306	9	Addition to list of superclasses	13,497
10	Constant field removed	12,979	10	Method no longer final	9202

7.2.1.2 License Violations

While dependency management tools such as RubyGems[80], Maven[81], or CocoaPods[82] have been introduced to automate the downloading and importing of libraries into projects, these libraries still originate from various authors and come with a plethora of OSS licenses (horizontal increase). One library can utilize another library, leading to hierarchies of libraries and license dependencies. All of these libraries' licenses must be compatible and compliant with each other. License violations and incompatibilities are an often-overlooked factor when recommending licenses and therefore can significantly impact the trustworthiness of software systems. When incompatible licenses are used together, a license violation occurs. A license violation is defined

[80] https://rubygems.org/

[81] search.maven.org

[82] https://cocoapods.org/

as “the act of making use of a (licensed) work in a way that violates the rights expressed by the original creator” [243]. That is, not following the legal terms and conditions set out in the source license. Software authors who commit a license violation open themselves up to the possibility of being sued; sometimes this risk can amount to millions of dollars.

7.2.2 Evolvable Quality Assessment Metamodel (SE-EQUAM)

Quality is a widely-used term to evaluate the maturity of development processes within an organization. Defining quality allows organizations to specify and determine if a product has met certain non-functional and functional requirements. However, as Kitchenham [244] states: “*quality is hard to define, impossible to measure, easy to recognize.*” Unlike functional requirements, where a single analysis technique (e.g., use case modeling) is sufficient to identify essentially all requirements, the same analysis is not appropriate for all quality requirements. Quality, as defined by ISO 9000:2000 [244], is the “*degree to which a set of inherent characteristics fulfills requirements*”, where a requirement is a “*need or expectation that is stated, generally implied or obligatory*”.

Assessing the evolvability of software systems has been addressed in existing research through the introduction of software quality models, e.g., McCall [245], ISO/IEC 9126[83], and QUALOSS [246]. These models share a common, while informal (not machine-readable), structural representation of software qualities (Figure 45).

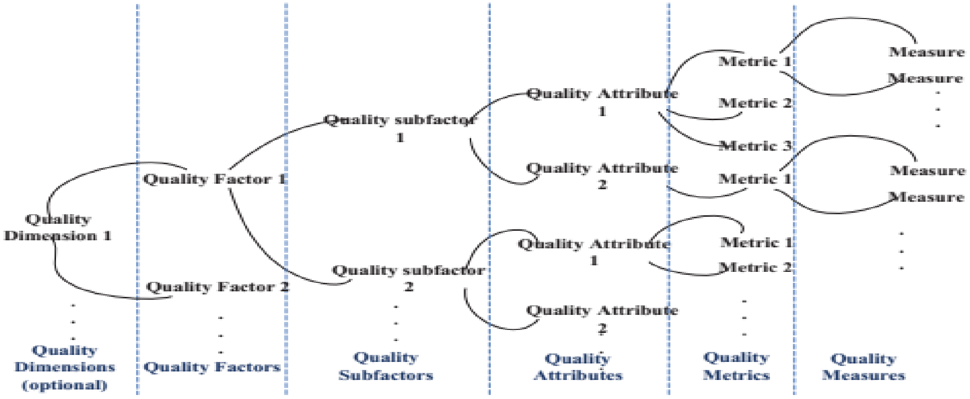


Figure 45: Generic structure of quality assessment models[244].

[83] https://www.iso.org/obp/ui/#iso:std:39752:en

While these models are capable of assessing qualities in a given context, they lack the required formalism and semantics to allow them to evolve to meet the modeling requirements of different assessment contexts. The ability to adjust to change assessment needs was a main motivation for SE-EQUAM, an Evolvable QUAlity Meta-model that derives a formal (machine-readable) domain model that can adapt to changes in the assessment needs in terms of both: artifacts being assessed and their assessment criteria [231]. SE-EQUAM addresses these challenges by taking advantage of the SW and its supporting technologies. SE-EQAM uses ontologies to model and conceptualize quality factors, sub-factors, attributes, measures, weights, and relationships used to assess software quality. Input artifacts for the assessment model are various software artifacts such as version control system and issue tracker; and its outputs are quality assessment scores based on the different assessment criteria. Ontologies not only provide a formal way to represent knowledge but can also eliminate ambiguity, enable validation, and provide a consistency-checking approach [177]. SE-EQUAM uses semantic reasoners to infer hidden relationships between domain model attributes. Given its formal representation SE-EQUAM allows for its reuse by simplifying the instantiation of new domain-specific instances of the model. More details about semantic reasoning are provided in [231].

Figure 46 illustrates the reuse and instantiation of our SE-EQAM model. The generic **syntactic meta-model,** which is a generic model that forms the basis for all quality models, can be instantiated by a domain model (e.g., ISO/IEC 9126). Furthermore, SE-EQUAM allows for a semantic mapping between the syntactical meta-model and a **semantic ontology meta-model**, which can then be instantiated as domain model ontology based on user-defined assessment criteria.

The SE-EQUAM Process. The general SE-EQUAM process (Figure 46) represents a set of tasks and activities which we followed to allow for deriving a generic quality assessment method that can be used to customize and instantiate the generic model to meet a stakeholder's specific quality assessment context.

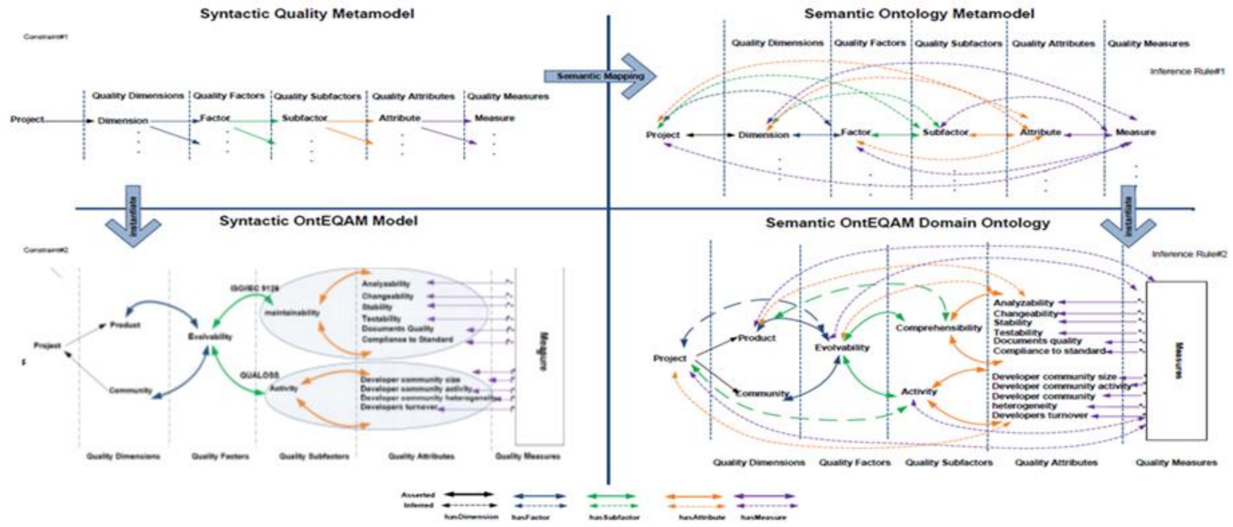


Figure 46: SE-EQUAM ontology meta-model reuse to instantiate a domain model ontology (OntEQAM)[231].

The inputs to the SE-EQUAM process are software artifacts and a set of core quality measurements applicable for these artifacts. In the next step, a common ontological representation for these artifacts has been established by re-using existing models or customizing existing models to meet the requirements of these artifacts. As part of the model adjustment activity, quality metrics and measurements included in the core model can be customized and extended to reflect a specific model context. The output of this process is an instantiated assessment model which meets specific user and project assessment requirements by providing quality assessment at both individual artifact and overall product levels. Figure 47 illustrates the high-level activities and major tasks involved in the SE-EQUAM instantiation method.

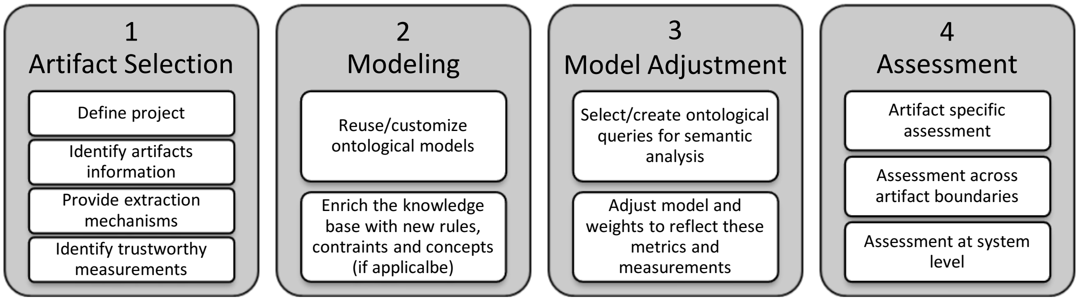

1
Artifact Selection
Define project
Identify artifacts information
Provide extraction mechanisms
Identify trustworthy measurements
2
Modeling
Reuse/customize ontological models
Enrich the knowledge base with new rules, contraints and concepts (if applicalbe)
3
Model Adjustment
Select/create ontological queries for semantic analysis
Adjust model and weights to reflect these metrics and measurements
4
Assessment
Artifact specific assessment
Assessment across artifact boundaries
Assessment at system level

Figure 47: SE-EQUAM process to instantiate evolvability model.

In the next section we introduce OntTAM, which illustrates a concrete instantiation of the SE-EQUAM process to create a semantic enriched trustworthiness quality assessment model for software libraries.

7.3 Ontology-based Trustworthiness Assessment Model (OntTAM)

OntTAM, an instantiation of the SE-EQUAM [231] ontology meta-model, illustrates how our modeling approach can take advantage of the unified ontological representation of both software artifacts and the generic SE-EQUAM quality assessment model. OntTAM instantiates a domain specific quality model to assess the trustworthiness of software projects and, more specifically, the trustworthiness of external libraries. OntTAM reuses SE-EQUAM's core quality model structure which is based on quality factors, sub-factors, attributes, measures, weights, and relationships, and extends them with trustworthiness specific aspects. Inputs to OntTAM are knowledge resources such as: version control systems, build systems, project license information, and security vulnerabilities information. The output of OntTAM is a trustworthiness assessment score for either an individual metric or an aggregation of sub-factors and factors for the overall product/library quality. The model thereby takes advantage of the OWL[84] and RDF/RDFS[85] semantic reasoning capabilities to infer hidden relationships between domain model attributes and to ensure consistency among these attributes.

Figure 48 provides an overview of the knowledge model framework and its organization in terms of ontologies and their abstraction levels. While these ontologies may be derived, modeled, and used independently, a key objective of our approach is knowledge integration across ontology boundaries, using both ontology alignments and semantic linking to create a unified ontological knowledge representation.

[84] https://www.w3.org/OWL/

[85] https://www.w3.org/TR/rdf-schema/

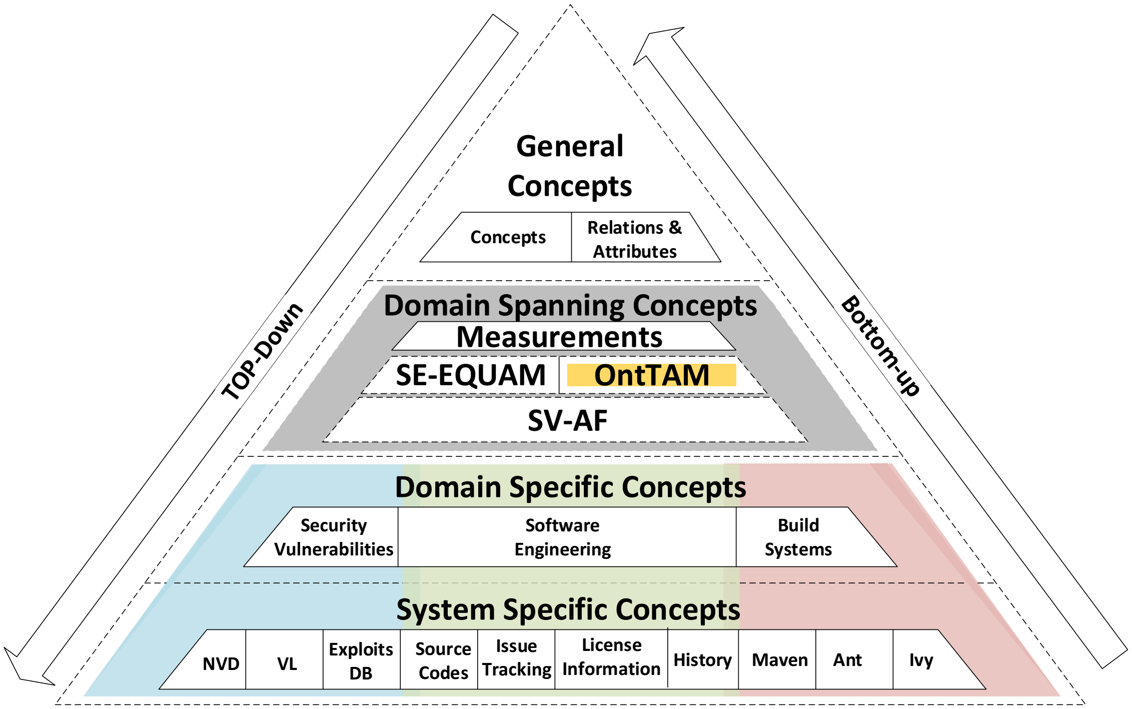


Figure 48: The Software Security and Trustworthy Ontology Hierarchy.

In what follows, we present our OntTAM methodology to further demonstrate how we instantiate different trustworthiness sub-factors (i.e., security, reliability, and legality) to establish a trustworthiness assessment for OSS products (e.g., external libraries). More specifically, we discuss in detail the four major steps involved in instantiating our customized OntTAM trustworthiness assessment model (Figure 47): artifact selection, modeling, model adjustment, and the assessment process.

7.3.1 Artifact Selection

The inputs to OntTAM are artifacts relevant to the reuse of software libraries within projects. These software artifacts can be categorized into endogenous and exogenous data (discussed in Chapter 5, section 5.3.2). Extracting and populating facts from these artifacts is often based on techniques commonly used by the MSR community [247]–[249]. It should be noted that unstructured or semi-structured information (e.g., vulnerability descriptions and license information) often requires several preprocessing steps such as natural language analysis (NLP), as well as data cleansing to improve the quality of the data prior to the ontology population. More details about our data preprocessing and ontology population process can be found in Chapters 5 and 6.

7.3.2 Model and Model Adjustment

In this section, we discuss our knowledge modeling process in detail. It should be noted that in order to improve readability, we will be using the following prefixes as substitutes for the fully-qualified names of our ontologies:

- rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
- owl: <http://www.w3.org/2002/07/owl#>
- seon: <http://se-on.org/ontologies/general/2012/02/main.owl#>
- sevont: <http://aseg.cs.concordia.ca/segps/ontologies/domain-spanning/2015/02/vulnerabilities.owl#>
- sequam: <http://aseg.cs.concordia.ca/segps/ontologies/domain-spanning/2017/09/sequam.owl#>
- onttam: <http://aseg.cs.concordia.ca/segps/ontologies/domain-spanning/2017/09/onttam.owl#>
- sbson: <http://aseg.cs.concordia.ca/segps/ontologies/domain-specific/2015/02/build.owl#>
- code: <http://aseg.cs.concordia.ca/segps/ontologies/domain-specific/2015/02/code.owl#>
- oswaldo: <http://aseg.cs.concordia.ca/segps/ontologies/domain-specific/2017/09/license.owl#>

7.3.2.1 Modeling Project Trustworthiness

Since OntTAM is based on the generic SE-EQUAM model, OntTAM is an extension and specialization of our core SE-EQUAM software quality assessment model. OntTAM is extended to provide a syntactical trustworthiness quality model that includes and defines a set of sub-factors, attributes, and metrics required for the assessment of trustworthiness. Many of these trustworthiness factors, attributes, and metrics are derived from existing work on trustworthiness assessment of open and closed source projects [231], [244]. The OntTAM specific trustworthiness assessment is based on the two general quality dimensions, the community and product dimension. The community dimension assesses the adoption of a software product by the community over an extended period of time by considering the popularity in terms of downloads, rankings, and activity of the development community. The product dimension assesses the internal structure of the product and the development processes that impact its reusability, which is the focus for this paper.

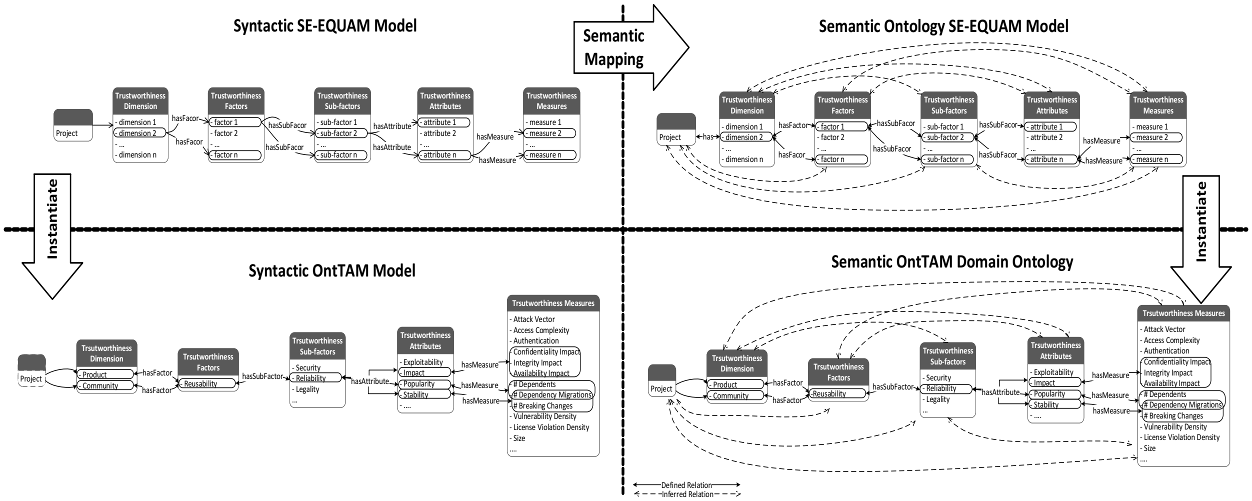


Figure 49: Reuse of the SE-QUAM meta-model to instantiate the OntTAM domain model ontology.

Figure 49 provides an overview of the complete model instantiation process which provides as its output a formal (machine redable) and semantic enriched trustworthiness assessment model. The process involves applying both a syntactic and semantic mapping from SE-EQUAM to OntTAM. While the syntactical model allows us to answer basic queries such as: *What are the sub-factors associated with product trustworthiness?* The semantic mapping enables the use of DL axioms (such as the property chain axiom) to infer new implicit relationships (dashed lines in Figure 49 – semantic OntTAM model) from explicitly modeled relationships in OntTAM (solid lines in Figure 49).

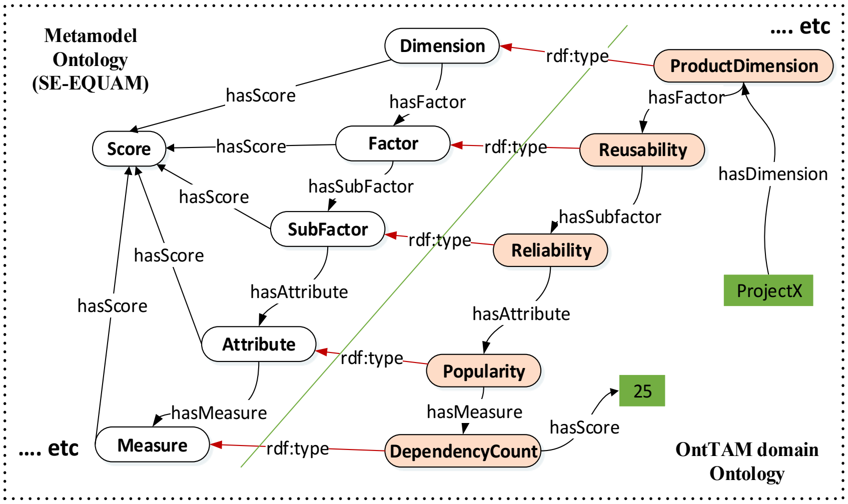


Figure 50: An example defining the associated trustworthiness concepts and measures for a sample project.

Figure 50 illustrates the main steps which are applied to associated trustworthiness concepts and measures for a sample project (ProjectX):

1. Define the *product* and *community* dimensions.
   <onttam:ProductDimension><rdfs:type><sequam:Dimension> and
   <onttam:CommunityDimension><rdfs:type><sequam:Dimension>.
2. Define *reusability* as a factor that is associated with the *product* dimension.
   <onttam:ProductDimension><sequam:hasFactor><onttam:Reusability> and
   <onttam:Reusability><rdfs:type><sequam:Factor>.
3. Following the same approach, OntTAM defines *reliability* as a sub-factor of *reusability* which is associated with the *popularity* attribute.
   <onttam:Reusability><sequam:hasSubfactor><onttam:Reliability>,
   <onttam:Reliability><rdfs:type><sequam:Subfacor>,
   <onttam:Reliability><sequam:hasAttribute><onttam:Popularity> and
   <onttam:Popularity><rdfs:type><sequam:Attribute>.
4. Assuming that OntTAM assesses a product's reusability through the *popularity* trustworthy attribute using the *DependencyCount* measure, we can now define this as:
   <onttam:Popularity><seon:hasMeasure><sbson:DependencyCount>and
   <sbson:DependencyCount><rdfs:type><seon:Measure>.

Finally, we enrich OntTAM's syntactical model to become a semantic model by establishing additional semantic relationships by adding property chain axioms (e.g., hasDimension relationship with hasSubfactor and hasMeasure). The following are examples of OWL 2 property chain axioms which we added to be able to take advantage of RDFS reasoning during the assessment process.

- Project-related OWL 2 property chain constructs:
    - SubPropertyOf(ObjectPropertyChain(:Project :hasFactor) :Factor)
    - SubPropertyOf(ObjectPropertyChain(:Project :hasSubfactor) :Subfactor)
    - SubPropertyOf(ObjectPropertyChain(:Project :hasAttribute) :Attribute)
    - SubPropertyOf(ObjectPropertyChain(:Project :hasMeasure) :Measure)
- Dimension-related OWL 2 property chain constructs:
    - SubPropertyOf(ObjectPropertyChain(:Dimension :hasSubfactor) :Subfactor)
    - SubPropertyOf(ObjectPropertyChain(:Dimension :hasAttribute) :Attribute)
    - SubPropertyOf(ObjectPropertyChain(:Dimension :hasMeasure) :Measure)
- Factor-related OWL 2 property chain constructs:
    - SubPropertyOf(ObjectPropertyChain(:Factor :hasAttribute) :Attribute)
    - SubPropertyOf(ObjectPropertyChain(:Factor :hasMeasure) :Measure)
- Subfactor-related OWL 2 property chain constructs:
    - SubPropertyOf(ObjectPropertyChain(:Subfactor :hasMeasure) :Measure)

7.3.2.2 Integration with Other Knowledge Artifacts

Assessing the overall trustworthiness of a software library requires us not only to instantiate OntTAM but also to integrate it with other ontological software knowledge artifacts to be able to derive and integrate novel trustworthiness measures. For the integration we take advantage of software artifact ontologies we have created and refined over the years [166], [250], [251], and reuse existing ontologies [20] that model different software artifacts. Figure 51 provides an overview of the unified ontological representation of software artifacts which we integrate with OntTAM. These artifacts include, but are not limited to: (a) Software Evolution Ontologies (SEON) which model SE repositories such as source code, version control systems, issue tracker systems, licenses, etc.; (b) the Build Systems ONtology (SBSON) which captures knowledge about build management systems (e.g., Maven); and (c) the SEVONT for modeling software security vulnerability information such as severities, impacts, vulnerabilities types, and patch information found in different security databases.

The integration of these heterogeneous knowledge resources allows us to introduce different trustworthiness measures related to the reuse of software libraries. More specifically, in this research we introduce the following three trust criteria: API breaking changes, security vulnerabilities, and license violations. Figure 51 shows the core concepts and object properties, distributed across the different abstraction layers of our knowledge modeling framework (Figure 48). It should be noted that we omitted data properties to improve readability of the figure.

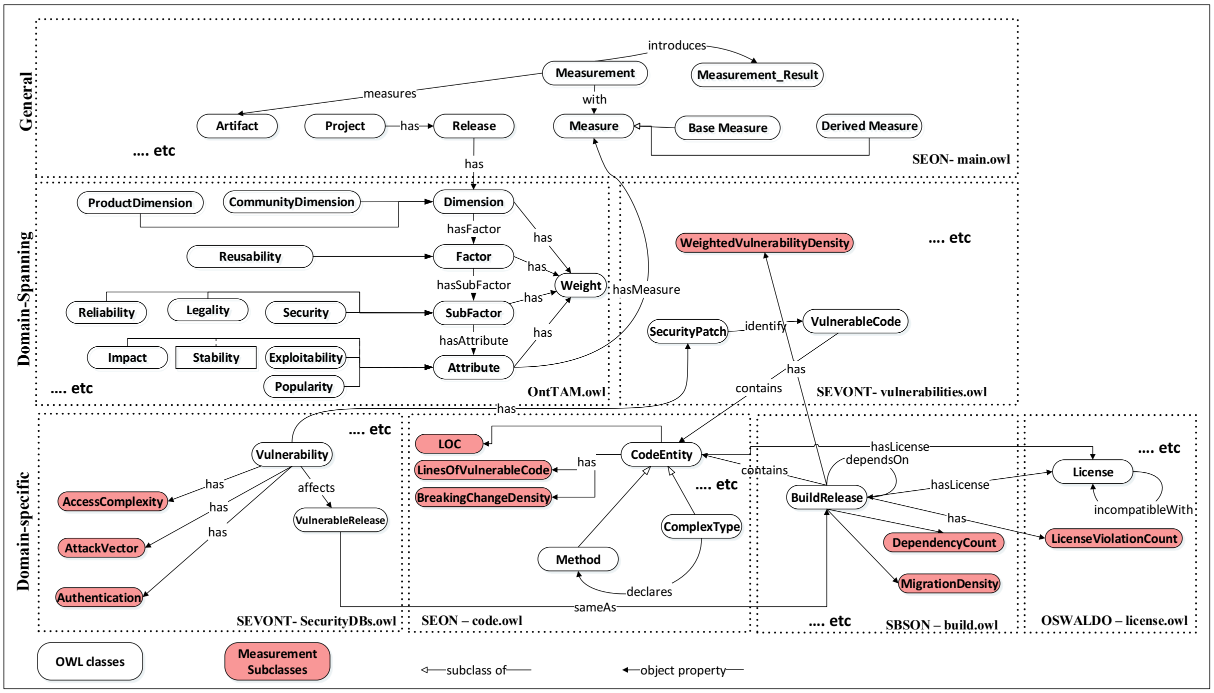


Figure 51: Integrating OntTAM ontology into SV-AF model and reusing SE-QUAM concepts.

Among the core concepts used from these ontologies are the BuildRelease from the SBSON build ontology, which is a subclass of the Release concept that also captures the fact that a project can have several releases (including library releases). A Release has a License, and defines its dependencies on other releases. Each release contains a set of CodeEntity elements such as Field, Method, and Class. A release can be affected by a Vulnerability, leading to the release of a new version containing a SecurityPatch. A security patch corresponds to code changes introduced to fix some existing VulnerableCode, which is part of a CodeEntity. 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 an object property. For a complete description of the ontologies, how they are built, the alignment processes, and reasoning, we refer the reader to Chapters 5 and 6.

All of these core concepts have metrics used by the OntTAM assessment process. Measures have a *unit* and are expressed on a *scale*, e.g., an ordinal or nominal scale. Information about units and scales can be used to perform conversions [252]. Many base measures, such as number of lines of vulnerable code (LOVC), number of known vulnerabilities, vulnerabilities severities (scores), and number of license violations, provide only limited insights when viewed in isolation. Additional derived measures are needed to support further analysis and assessment of software artifacts. These derived measures represent an aggregation of values from different subdomains, for example, the number of vulnerabilities per class is an aggregation of measures derived from source code and the vulnerability repositories. While the abstract measurement concepts are defined in the general upper layer of our integrated model (Figure 51), many **Base Measures** (e.g., Size) and **Derived Measures** (e.g., Weighted Vulnerability Density) are modeled in the domain-specific layer.

7.3.3 Measures and Metrics

An essential feature of our modeling approach is to allow users to customize the OntTAM model through user defined queries which might introduce different metrics, ranging from simple metrics to semantic rich metrics queries that take advantage of implicit knowledge inferred by ontological reasoners. Given our ontology based modeling approach, these analysis results can also be materialized to enrich our knowledge base and to promote reuse of existing analysis results. Next, we introduce some metrics to be used later for the assessment of the trustworthiness of systems. These metrics take advantage of not only our unified representation, but also inference services provided by the SW.

Weighted Vulnerability Density (WVD) Metric compares software systems (or their components) based on severity scores of known vulnerabilities. The objective of WVD is to measure the impact of known vulnerabilities on a product's quality, with the most severe vulnerabilities having the greatest impact. The metric can be applied, for example, to prioritize the patching of vulnerabilities based on their severity. To account for both direct and indirect impacts of vulnerabilities, we introduce the $\text{WVD}_{\text{direct}}$ and $\text{WVD}_{\text{inherit}}$ measures. Although a project can have a $\text{WVD}_{\text{direct}}$ score of 0 since no known security vulnerability has been reported for the core project, it is still possible that the project is exposed to indirect vulnerability found in external (third party) dependencies (components) that are included in the parent project. Such a potential security risk will be assessed by the $\text{WVD}_{\text{inherit}}$ measure.

$$WVD_{direct}(release) = \frac{\sum_{i=1}^{V} w_i}{S} \tag{2}$$

where S is the size of the software (in KLOC), w_i is the weight (severity score) of a known vulnerability affecting the system, and V is the number of known vulnerabilities in the system.

$$WVD_{inherit}(release) = \sum_{i=1}^{n} \left\{ \left(\frac{vulnerable\ APIs\ in\ d_i\ used\ by\ release}{total\ vulnerable\ APIs\ in\ d_i} \right) * WVD_{direct}(d_i) \right\} \tag{3}$$

where n is the number of dependencies used by release, and d_i is the i^th dependency.

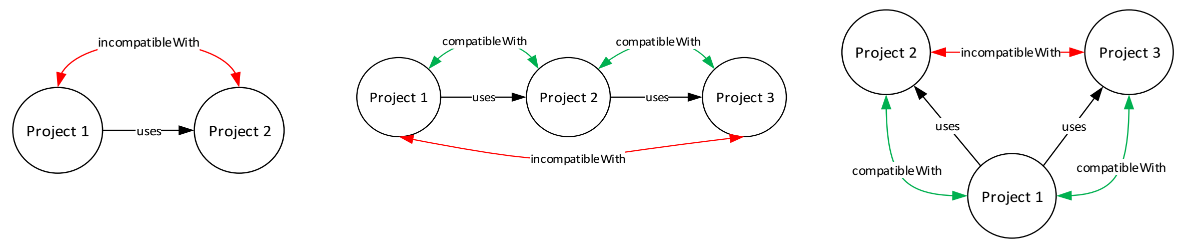
$$WVD_{overall}(release) = WVD_{direct}(release) + WVD_{inherit}(release) \tag{4}$$

License Violation Count (LVC) is a measure that assesses the number of license violations that exist within a given project. This measure can indicate potential long-term risks associated with intellectual rights violations that exist within a project. A license violation occurs if any of the dependent components of a parent project include components with non-compatible licenses. Open source code license violations are often due to the fact that many software developers are simply neither aware nor well-versed in open source license compliance. For example, in 2008 the Free Software Foundation (FSF) claimed that various products sold by Cisco under the Linksys brand had violated the licensing terms of many programs on which FSF held copyright[86]. These FSF programs were under the GNU General Public License, a copyleft license which allows users to modify a piece of software as long as the derivative work is under the same license.

[86] https://en.wikipedia.org/wiki/Free_Software_Foundation,_Inc._v._Cisco_Systems,_Inc.

In this work, we identify three main categories of license violations: *simple violations*, *transitive violations*, and *compound violations* (see Figure 52). LVC$_{\text{simple,}}$ LVC$_{\text{transitive,}}$ and LVC$_{\text{compound}}$ are base measures associated with each category. Details on how license violations are identified are presented in Section 7.4.3.

$$LVC_{overall}(release) = LVC_{simple}(release) + LVC_{transitive}(release) + LVC_{compound} \tag{5}$$



(a) Type 1 - Simple Violations

(b) Type 2 - Transitive Violations

(c) Type 3 - Compound Violations

Figure 52: Categories of license violations.

Breaking Change Density (BCD) Metric is a normalized measure which represents the ratio between breaking and non-breaking API changes that are introduced in a project. API changes often occur as a project and its components evolve inconsistently, resulting in incompatibilities of APIs and API calls. This measure can be used to determine the stability of an API over time—how often do breaking changes occur. Details on how we identify breaking changes are presented in Section 7.4.4. The BCD metric can be represented formally as follows:

$$BCD = \frac{\#\ breaking\ API\ changes}{\#\ nonbreaking\ API\ Changes} \tag{6}$$

Breaking Change Impact (BCI) measures the impact of breaking changes on client applications by assessing a client application and its use of APIs with a changed contract. The impact of breaking changes on clients can be both direct and indirect. While there exists a significant body of work on the direct impact of changes ([253], [237], [242], [254], [255]), very little research has been conducted on indirect breaking changes. Indirect breaking changes occur, for example, when different versions of the same API are introduced by any of the client's other dependencies. By default, the Java Virtual Machine is unable to differentiate between multiple versions of the same API. In cases where multiple versions of a dependency are encountered, the first occurrence of an API version in a project's class-path is chosen. We introduce two measures that capture both direct and indirect breaking changes.

We represent the BCI metrics formally as follows:

$$BCI_{direct}(C, D) = \frac{\#\ breaking\ API\ changes\ in\ D\ used\ by\ C}{\#\ breaking\ API\ Changes\ in\ D} \tag{7}$$

$$BCI_{indirect}(C, < D_1, \dots ,\ D_n >) = \frac{\#\ breaking\ API\ changes\ in < D_1, \dots , D_n > used\ by\ C}{\#\ breaking\ API\ Changes\ across < D_1, \dots , D_n >} \tag{8}$$

where C is the client project, D is the reused library, and <D_1, …, D_n> is the set of (direct and transitive) different library releases being reused by the client.

7.3.4 Assessment Process

Given that assessment needs differ among stakeholders and assessment contexts, our OntTAM assessment process allows for the customization of trustworthiness assessment model in terms of sub-factors and attributes being assessed as well as the individual weights assigned to them. While the default weight for all sub-factors and attributes are equal, users can customize these weights to more closely match their assessment objective and context. Furthermore, while most existing assessment approaches rely on crisp boundaries (e.g., based on thresholds), this approach can lead to inaccuracy in the assessment process. It is not always feasible or desirable to use crisp values, especially when one deals with values which are close to crisp value boundaries. For example, let us assume a project X with a reported number of 5 known vulnerabilities, a binary scale for trustworthiness which is trustworthy or non-trustworthy, and a crisp value threshold of 4 known vulnerabilities. Based on this crisp boundary, the project will be assessed as being non-trustworthy, even if it can be considered almost borderline trustworthy. To further exemplify the problem, by using the crisp boundary values there would not be any difference between project X with 5 known vulnerabilities and project Y with 100 vulnerabilities; both projects would be considered equally non-trustworthy. Furthermore, the problem can occur not only at the individual measurement level but also at other assessment levels (e.g., sub-factor, factor). To address this challenge we apply a fuzzy logic assessment and inference approach to eliminate the need for crisp value boundaries.

Figure 53 shows the set of transformation steps which are performed during the fuzzification of the assessment process, with details of each step discussed more thoroughly throughout the section.

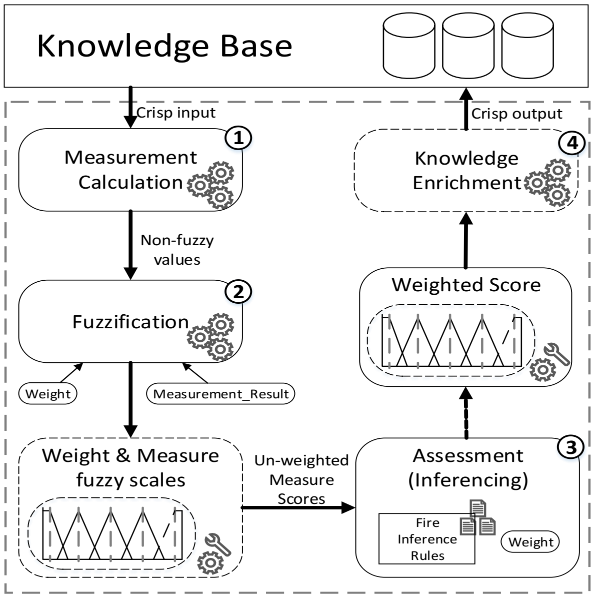


Figure 53: Fuzzy Assessment Process Steps.

***(1) Measure Calculation*:** Inputs to this step are raw values from the populated ontologies. Measures are calculated by querying our populated knowledge base for the base and derived measures introduced in the previous section (e.g., *WVD*).

(2) Fuzzification: The extracted quality measures and weight values are used to create fuzzy scales in the fuzzification step. As part of the fuzzification step, fuzzy scales are created for the different measures, the assessment weights (provided by stakeholders of the assessment model to assign a level of importance to different measures), and the overall assessment result. These results are converted to linguistic variables, which are variables whose values are expressed as words or sentences (e.g., high, not very high, low) [256]. These linguistic variables are the building blocks of Fuzzy Logic and become the input for the fuzzification inference engine.

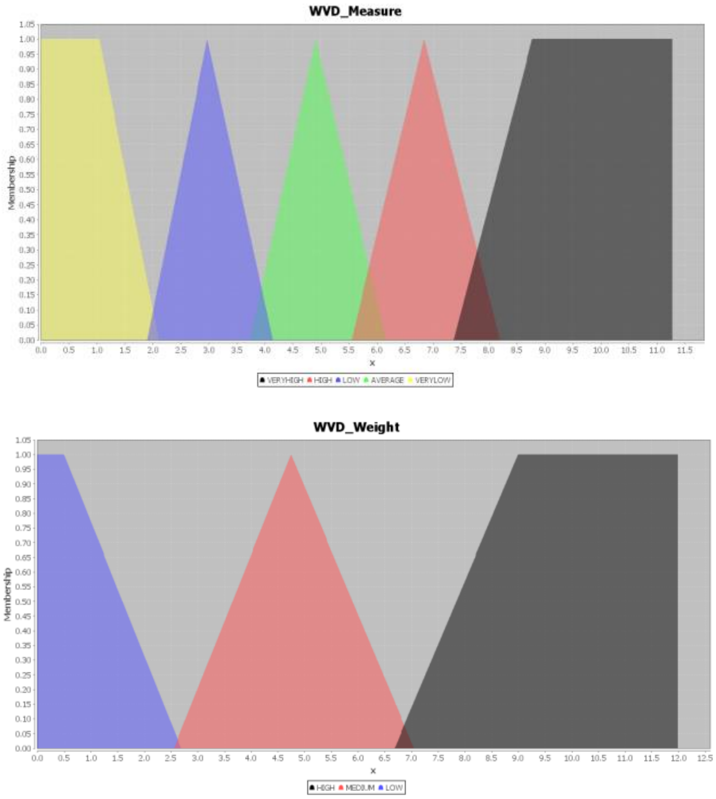


Figure 54: WVD measure fuzzy scale and Weight Fuzzy Scale for WVD measure.

Figure 54 shows an example of a fuzzy scale created for the WVD measure and its assessment weights. The x-axis represents the measurement results range and the y-axis the membership degree (range is 0–1). The higher the membership value, the stronger the measurement's relation to its fuzzy result scales. The overlap between boundaries of categories in the fuzzy scale demonstrates the uncertainty in interpreting boundary measurement results.

Since high WVD, LVC, and BCD measures lower the overall quality and trustworthiness score of a project, we made the following three assumptions to automate the fuzzy inference rules for these measures: (1) In cases when the user-specified weight is high then the individual measure score is one level lower. *VeryPoor* scores will keep their values (e.g., a high weight will change an *Excellent* score to *VeryGood*). (2) The opposite holds for low weights, which reflects that their scores are less relevant to the overall assessment, their scores are adjusted by one level higher. *Excellent* scores keep their values. (3) With medium weights, scores keep their values.

These assumptions reflect the fact that when a measure is of high importance to the assessment (high weight), its score should be more sensitive to a low measure value.

(3) Inference and Assessment: Input for this step is the fuzzified measure and weight values in the form of linguistic variables. These linguistic results are now transformed into the final assessment score by executing a set of fuzzy inference rules. The de-fuzzification is based on a set of fuzzy inference rules, which are expressed in the Fuzzy Control Language (FCL)[257] using the JFuzzyLogic inferencing engine [258]. The inference engine fires the relevant fuzzy rules based on the provided input. Firing rules will calculate the final weighted overall measurement result which is a combination of all the different measures. Using the Center of Gravity (COG) method, considered one of the most popular de-fuzzification methods [259], the overall fuzzy measurement result is de-fuzzified back into a numerical assessment measurement result in order to be populated back to the knowledge base. As part of our assessment, we created a Fuzzy Control Language (FCL) file for each measure. The complete set of FCL files for all measures can be found online[87].

***(4) Knowledge Enrichment*:** This optional step allows for the integration of the assessment results at the individual attribute, sub-factor, and overall assessment levels. Our ontological representation enables us to seamlessly integrate these assessment results into the knowledge base, therefore not only supporting reuse of analysis results but also allowing their use for further semantic analysis.

7.4 Case Study

The objective of this section is to demonstrate the applicability of our modeling approach to support the assessment of trust within OSS software libraries by highlighting the flexibility of our modeling approach, in terms of its seamless knowledge and analysis results integration, as well as the use of SW reasoning services to infer new knowledge (measures). In Section 7.4.1, we present the setup for our study, including the selection process for the 4 projects used to illustrate our approach; Sections 7.4.2 to 7.4.4 describe how we identify and measure security

[87] https://github.com/segps/segps-code

vulnerabilities, license violations, and API breaking changes. Section 7.4.5 describes how these individual identified measures can be integrated for a holistic trustworthiness assessment.

7.4.1 Study Setup

For the data collection and extraction in our case study (see Figure 55), we rely on four data sources: the NVD database, GitHub, SVN, and the Maven build repository.

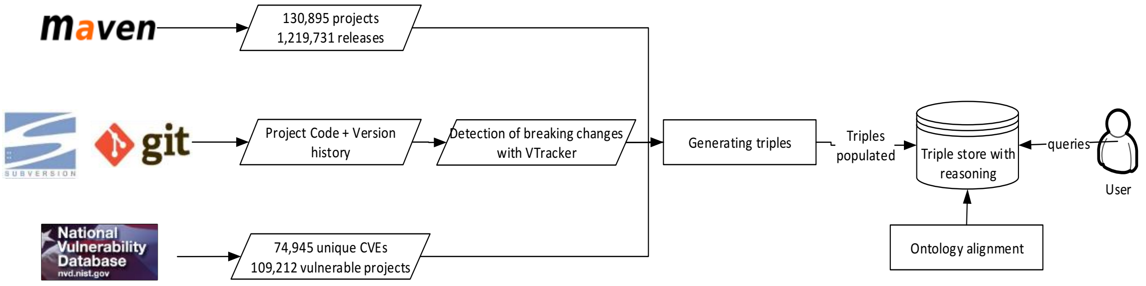

maven
130,895 projects
1,219,731 releases
SUBVERSION
git
Project Code + Version history
Detection of breaking changes with VTracker
National Vulnerability Database
nvd.nist.gov
74,945 unique CVEs
109,212 vulnerable projects
Generating triples
Triples populated
Triple store with reasoning
queries
User
Ontology alignment

Figure 55: Overview of case study setup process.

For our study we downloaded the latest versions of the Maven and NVD repositories—which include 1,219,731 project releases in Maven and 74,945 vulnerabilities affecting 109,212 releases in NVD. For our study, we limited the assessment scope to 4 projects. The projects were selected based on the following criteria: (a) at least some of their releases contained known vulnerabilities, (b) license details were provided, (c) releases varied in their major version numbers, and (d) the functionalities these products provide are widely reused by other projects (see Table 43 for details). The four subject systems vary in size (classes and methods) and application domain. Commons Fileupload[88] adds file upload capabilities to web applications. CXF WS Security[89] provides reusable components for client-side authentication, security, and encryption. Struts[90] is an open source framework for creating Java web applications, and ASM[91] is a Java bytecode manipulation library. We further extract the complete source code and history information of these four projects. The extracts facts used to populate the corresponding ontologies and made persistent in our triple store.

[88] https://commons.apache.org/proper/commons-fileupload/
[89] http://cxf.apache.org/docs/ws-security.html
[90] https://struts.apache.org/
[91] http://asm.ow2.org/

Table 43: Overview of selected case study projects

Project	**# Releases analyzed**	**# of Dependencies**
Commons Fileupload	6	68854
Apache CXF WS Security	5	4570
Struts	3	3170
ASM	20	8109

7.4.2 Identifying and Measuring Software Security Vulnerabilities

Approach. In what follows, we show some of the main rules and queries used to derive the WVD measures (overall, direct, and inherited). These rules are of interest since they highlight the flexibility and power of our modeling approach, allowing users to define and customize their own derived measures without the need for any additional proprietary algorithms implementations or modelling.

$\underline{\text{WVD}_{\text{direct}}\text{ inference}}$: In order to derive the $\text{WVD}_{\text{direct}}$ score for the projects, we define rules using the Semantic Web Rule Language (SWRL), similar to the one shown in Listing 14. The rule states that if some project release has a LOC and OverallSeverityScore measure, then the release has a $\text{WVD}_{\text{direct}}$ score (obtained by dividing the overall severity score by LOC).

```
Release(?r), hasLOC(?r, ?loc), hasOverallSeverityScore(?r, ?score),
divide(?wvdDirect, ?score, ?loc) → hasDirectWVD(?r, ?wvdDirect)
```

Listing 14: The rules to infer the direct WVD measure.

$\underline{\text{WVD}_{\text{inherit}}\text{ inference}}$: For us to be able to infer the $\text{WVD}_{\text{inherit}}$ measure of a project release, we had to first determine the raio of vulnerable APIs that are reused in a particular release. The OntTAM knowledge model not only captures the required information to derive this measure, but also includes all semantics to be able to take advantage of the SW reasoners to infer the measure value. More specifically, once the required ontologies (e.g., SEVONT, SEON, OntTAM) are populated, a SPARQL query can be created to retrieve the number of vulnerable API elements in a given release (see Listing 15).

```
CONSTRUCT{?release sevont:hasVulnerableCodeCount ?totalVulnerableCodeCount}
WHERE{
{
  SELECT ?release count(?vulnerableCode) as ?totalVulnerableCodeCount
  WHERE{
    ?vulnerableCode rdf:type code:VulnerableCode.
    ?release code:containsCodeEntity ?vulnerableCode
  }GROUP BY ?release
}}
```

Listing 15: SPARQL query for inferring the total number of vulnerable code entities in a project.

Using Listing 16 we can also determine the number of such vulnerable API elements being reused in client applications. For a more detailed description on how we detect vulnerable code elements, the reader is referred to Chapter 6, section 6.3.1.

```
CONSTRUCT{?link sevont:hasReusedVulnerableCodeCount ?usedVulnerableCodeCount}
WHERE{
{
  SELECT ?link count(?vulnerableCode) as ?usedVulnerableCodeCount
  WHERE {

   #?link represents the ?client dependency on ?release
   ?link a build:DependencyLink.
   ?link build:hasDependencySource ?client.
   ?link build:hasDependencyTarget ?release.
   ?client code:containsCodeEntity ?codeEntity.
   ?codeEntity main:dependsOn ?vulnerableCode.
   {
      SELECT ?vulnerableCode
      WHERE {
         ?vulnerableCode rdf:type code:VulnerableCode.
         ?release code:containsCodeEntity ?vulnerableCode.
        }
    }
  }GROUP BY ?link
}}
```

Listing 16: The SPARQL query for inferring the vulnerable code entities used by different dependent projects.

The SPARQL query (Listing 17) exemplifies how we take advantage of analysis results from the inference rules in Listing 14 to infer the final $\text{WVD}_{\text{inherit}}$ measure for a particular release of a component.

```
CONSTRUCT{?client sevont:hasInheritWVD ?inheritWVD }
WHERE{
{
  SELECT ?client count(?indirectWVD) as ?inheritWVD
  WHERE {
   ?link a build:DependencyLink.
   ?link build:hasDependencySource ?client.
   ?link build:hasDependencyTarget ?release.
   ?client sevont:hasReusedVulnerableCodeCount ?usedVulnerableCodeCount.
   ?release sevont:hasVulnerableCodeCount ?totalVulnerableCodeCount.
   ?release sevont:hasDirectWVD ?directWVD.
   BIND((?usedVulnerableCodeCount/?totalVulnerableCodeCount) AS ?vulnerableCodeRatio).
   BIND((?vulnerableCodeRatio * ?directWVD) AS ?indirectWVD).
  }
}}
```

Listing 17: SPARQL query for inferring inherited WVD measures in clients' projects.

Findings and Discussion. Table 44 provides the analysis results for our case study in terms of known vulnerabilities, size, and WVD scores for selected project releases. Using the WVD measure we can now compare two releases of the same project in terms of their weighted vulnerability density. For example, based on the WVD measure, we can consider Struts 1.2.9 to be more trustworthy than earlier versions of Struts (e.g., version 1.2.4 and 1.2.8, which both have higher WVD scores).

Table 44: Vulnerability densities of selected projects

Project	# vulnerabilities	Aggregated Vuln. Scores	Size (Kloc)	WVD
commons-fileupload 1.0	2	10.8	1.23	8.78
commons-fileupload 1.1	2	10.8	1.28	8.46
commons-fileupload 1.2	2	10.8	1.78	6.05
commons-fileupload 1.2.1	2	10.8	1.97	5.49
commons-fileupload 1.2.2	2	10.8	2.04	5.31
commons-fileupload 1.3	1	7.5	2.39	3.14
Apache CXF WS Security 2.4.1	4	23.6	18.92	1.25
Apache CXF WS Security 2.4.4	4	23.6	21.30	1.11
Apache CXF WS Security 2.4.6	5	27.9	23.10	1.21
Apache CXF WS Security 2.6.3	8	39.4	26.43	1.49
Apache CXF WS Security 2.7.0	10	49.4	26.43	1.87
Struts 1.2.4	5	30	24.04	1.25
Struts 1.2.8	8	49.6	24.61	2.02
Struts 1.2.9	4	25.7	24.76	1.04

We further analyzed the WVD results to see whether developers actually migrate their applications to library versions which are less vulnerable (e.g., a newer version of the same library with patched vulnerabilities). Table 45 provides an overview of the number of dependent applications which change their build dependency to a more trustworthy release (based on the lower WVD score). Our analysis results show that 45.1% of client applications switched their library dependencies and of these, 63.29% switched to a more trustworthy library release. Surprisingly, the remaining 36.71% switched to library releases which are either equal or less trustworthy (higher WVD score), even if more trustworthy library versions were available.

Table 45: Clients who switched from a vulnerable API in later release

Project	**Vulnerability**	**% clients switched versions of the library**	**% clients switched to less vulnerable release (WVD)**	**% clients switched to a release with equal or higher WVD score**
commons-fileupload 1.0	CVE-2014-0050	29.36%	74.26%	25.74%
commons-fileupload 1.1		6.28%	58.33%	41.67%
commons-fileupload 1.2		70.54%	100.00%	0.00%
commons-fileupload 1.2.1		38.97%	97.55%	2.45%
commons-fileupload 1.2.2		46.79%	99.99%	0.01%
commons-fileupload 1.3		40.62%	0.00%	100.00%
Apache CXF WS Security 2.4.1	CVE-2013-0239	94.93%	100.00%	0.00%
Apache CXF WS Security 2.4.4		95.00%	0.23%	99.77%
Apache CXF WS Security 2.4.6		95.24%	63.10%	36.90%
Apache CXF WS Security 2.6.3		98.08%	85.29%	14.71%
Apache CXF WS Security 2.7.0		92.75%	97.26%	2.74%
Struts 1.2.4	CVE-2016-1181	0.00%	n/a	n/a
Struts 1.2.8		44.44%	100.00%	0.00%
Struts 1.2.9		0.00%	n/a	n/a

7.4.3 Identifying and Measuring License Violations

Approach. License violations originating from external libraries and components can cause a major long-term liability for client applications in terms of intellectual property and the trustworthiness of these libraries. In our study we first evaluate if such license violations (non-compliances) occur in general in project dependencies managed by the Maven repository. In the second part of our study we revisit our 4 projects used in our trustworthiness assessment study to assess their trustworthiness in terms of license violations. For the study we created SPARQL queries that analyze all dependency relationships in Maven and identified three (3) main categories of license violations: *simple violations*, *transitive violations*, and *compound violations* (see Section 7.3.3). The queries take advantage of both our open source license ontology and the build ontology. Listings 18, 19, and 20 below illustrates the queries we used to identify these violations.

```
SELECT distinct *
WHERE {
 ?link a build:DependencyLink.
  ?link build:hasDependencyTarget ?project2.
  ?link build:hasDependencySource ?project1.
  ?project1 markosLicense:coveringLicense ?license1.
  ?project2 markosLicense:coveringLicense ?license2.
  ?license1 markosCopyright:incompatibleWith ?license2.
}
```

Listing 18: SPARQL query for inferring the total number of breaking changes in a project.

```
SELECT distinct *
WHERE {
 ?linkA a build:DependencyLink.
  ?linkA build:hasDependencyTarget ?project2.
  ?linkA build:hasDependencySource ?project1.
  ?linkB a build:DependencyLink.
  ?linkB build:hasDependencyTarget ?project3.
  ?linkB build:hasDependencySource ?project2.
  ?project1 markosLicense:coveringLicense ?license1.
  ?project2 markosLicense:coveringLicense ?license2.
  ?project3 markosLicense:coveringLicense ?license3.
  ?license1 markosCopyright:compatibleWith ?license2.
  ?license2 markosCopyright:compatibleWith ?license3.
  ?license1 markosCopyright:incompatibleWith ?license3.
}
```

Listing 19: SPARQL query for inferring the total number of breaking changes in a project.

```
SELECT distinct *
WHERE {
 ?linkA a build:DependencyLink.
  ?linkA build:hasDependencyTarget ?project2.
  ?linkA build:hasDependencySource ?project1.
  ?linkB a build:DependencyLink.
  ?linkB build:hasDependencyTarget ?project3.
  ?linkB build:hasDependencySource ?project1.
  ?project1 markosLicense:coveringLicense ?license1.
  ?project2 markosLicense:coveringLicense ?license2.
  ?project3 markosLicense:coveringLicense ?license3.
  ?license1 markosCopyright:compatibleWith ?license2.
  ?license1 markosCopyright:compatibleWith ?license3.
  ?license2 markosCopyright:incompatibleWith ?license3.
}
```

Listing 20: SPARQL query for inferring the total number of breaking changes in a project.

Findings and Discussion. This section presents and discusses the results obtained in our license violation experiment for the Maven repository. Figure 56 shows the distribution of common project licenses in the Maven repository. Table 46 reports on the license violations, classified by the type of violation, which we observed in our study of the Maven repository.

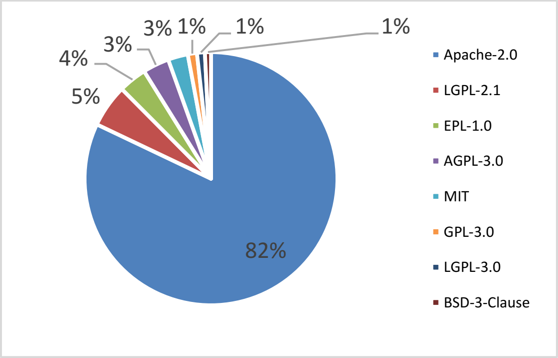


Figure 56: License distribution in the Maven repository.

Table 46: Totals of each type of violation found by querying the data store

License Violation Types	**Count**
Type 1 - Simple Violations	131 996
Type 2 - Embedded Violations	288 153
Type 3 - Compound Violations	654 964

Our study identified over 131,000 simple violations and numerous transitive license violations of various types. We note that Type 3 is seemingly the most popular type of violation, followed by Type 2, then Type 1. In what follows, we report on some of the license violations or incompatibilities which we observed in our study.

Figures 57, 58, and 59 summarize the most common license violation pairs which occurred for all three license violation categories. The most common, Type 1, violation which we observed is code published under the Apache 2 license being incorporated into GPL 2 licensed code. This violation is not surprising for two reasons. First, many software developers are simply not aware nor well-versed in open source license compliance, and as these are two of the most popular licenses in the world, this pairing reflects their usage in the wild. Second, there is likely some confusion about Apache 2's compatibility with the GPL. On the GNU website, the Free Software Foundation publishes a list of licenses that are compatible with the GPL. This page shows Apache 2 in green (meaning compatible), but in the license discussion the authors explain that Apache 2 is only compatible with GPL 3, not GPL 2 [240].

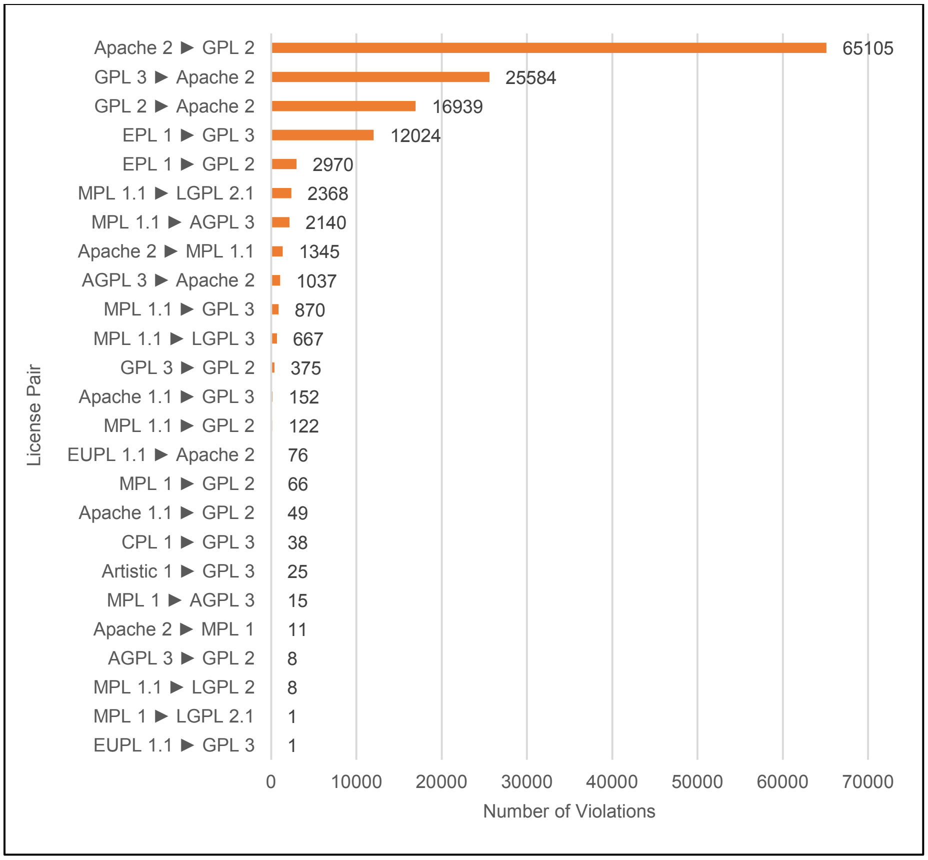


Figure 57: Most Popular Type 1 License Violation Pairs.

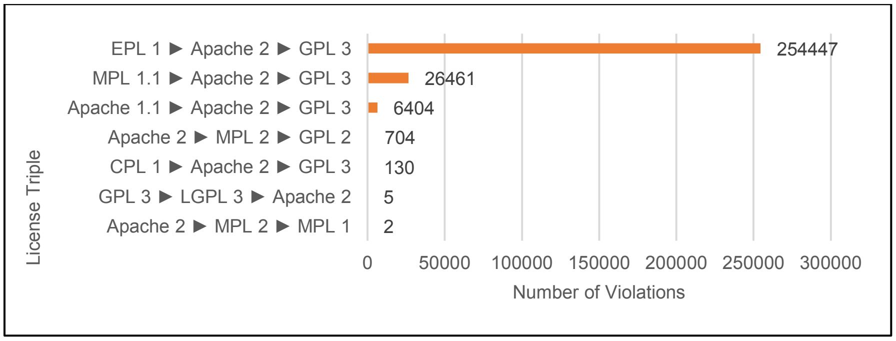


Figure 58: Most Popular Type 2 License Violation Pairs.

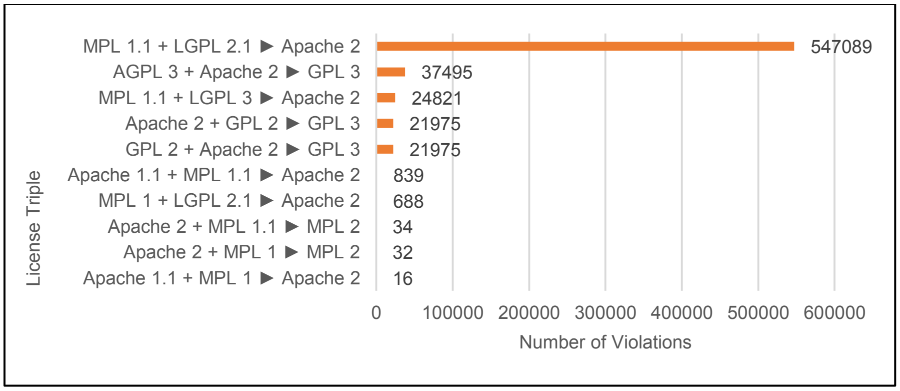


Figure 59: Most Popular Type 3 License Violation Pairs.

A more detailed analysis of the reasons why the number of transitive license violations is significantly larger compared to direct violations revealed: (1) Type 1 license compatibility/incompatibility are easier to verify/detect. That is, it is much more likely that a developer will check for license compliance, when only two licenses are involved. (2) Transitive violation types on the other hand, have not been considered in the research community prior to this work, and may very well be acceptable or be clearly identifiable as such. For example, the European Union Public License (EUPL) explicitly states which licenses it is compatible with. This is a known compatibility. Whereas for transitive interactions, the EUPL may then be imported into an intermediary project, say a project under the Licence Libre du Québec – Réciprocité (LiLiQ-R), which is then imported into a tertiary project under Common Development and Distribution License (CDDL). Each step (EUPL to LiLiQ, and LiLiQ to CDDL) is known to be compatible. But the EUPL does not explicitly state that it is compatible with the CDDL. This chain of licenses may be flagged as a violation by our approach. Yet this chain could in fact be perfectly lawful (a false-positive, verifiable by a lawyer). Our approach will however flag such a dependency chain as a potential violation. This triple is neither a known compatibility nor known incompatibility, and thus is one of the reasons why there are more Type 2 violations found.

Identification of Type 3 violations becomes even more difficult to detect since their detection largely depends on how licenses define derivative works and conditions for reusing these libraries. Libraries can be used by either including the actual source code or through linking (e.g. through a jar file). Linking of a library can be static (compile-time) or dynamic (run-time). For example, LGPL requires each project to be an "independent work that stands by itself, and includes no source code from [the other].” In this scenario, however, it is perfectly acceptable to combine compiled code [260]. So basically, the question is whether a derivative work is created or not when combining dependencies into a new project. Derivative works come into play only when the licensed software is copied, distributed, or modified. Additional research is needed to further clarify legal and license compliance issues when using these open source licenses. However, as can be noted, all three types of violations can exist in projects. Thus, simple, transitive, and complex license violations are a problem that occur in open source projects and can potentially affect the trustworthiness of components and libraries being reused in software projects.

Next, we report on license violations results which we observed for the selected 4 projects of our trustworthy study. Table 47 provides an overview of the number of license violations detected in these projects. Only four (4) releases of Commons-Fileupload introduced violations in client applications. No license violations are reported for the projects due to the lack of license information in the analyzed client applications. Results, although incomplete, confirm our previous claim that violations are problems that occur in open source projects.

Table 47: Licence Violation Counts in selected projects.

Project	# Simple Violations	# Transitive Violations	# Compound Violations
commons-fileupload 1.0	0	0	0
commons-fileupload 1.1	0	0	0
commons-fileupload 1.2	4	0	0
commons-fileupload 1.2.1	14	0	0
commons-fileupload 1.2.2	19	0	0
commons-fileupload 1.3	4	0	0
Apache CXF WS Security 2.4.1	0	0	0
Apache CXF WS Security 2.4.4	0	0	0
Apache CXF WS Security 2.4.6	0	0	0
Apache CXF WS Security 2.6.3	0	0	0
Apache CXF WS Security 2.7.0	0	0	0
Struts 1.2.4	0	0	0
Struts 1.2.8	0	0	0
Struts 1.2.9	0	0	0

7.4.4 Identifying and Measuring API Breaking Changes

Approach. As previously mentioned in our study setup (Section 7.4.1, Figure 55), we extract the source code and versioning information of the four projects from GitHub and SVN. For each successive pair of releases of a given project, we then identify the introduced breaking and non-breaking changes using the VTracker[92] tool. In order to be able to reuse the analysis results for further analysis, we take advantage of our ontological knowledge modeling approach and extend our knowledge base to include the analysis results. Developers can now access this information, using SPARQL queries, to derive potential direct and indirect impacts of breaking changes on their client applications. In what follows, we show some of the main rules and queries used to derive the BCD and BCI measures.

BCD inference: For computing the BCD scores of the projects in our dataset, we define a SWRL rule (see Listing 21) which infers the BCD score from the breaking and non-breaking change counts. Listings 22 and 23 detail the queries for computing the breaking and non-breaking change measures of a project.

```
Release(?r), hasBreakingChangeCount(?r, ?bcc),
hasNonBreakingChangeCount (?r, ?nbcc), divide(?bcd, ?bcc, ?nbcc)
→ hasBCD(?r, ?bcd)
```

Listing 21: The rules to infer the BCD measure.

```
CONSTRUCT{?release code:hasBreakingChangeCount ?totalBreakingChanges }
WHERE{
{
  SELECT ?release count(?breakingChange) as ?totalBreakingChanges
  WHERE{
    ?breakingChange rdf:type code:BreakingChange.

    ?breakingChange code:hasCurrentAPI ?api.
    ?release code:containsCodeEntity ?api.
  }GROUP BY ?release
}}
```

Listing 22: SPARQL query for inferring the total number of breaking changes in a project.

[92] https://users.encs.concordia.ca/~nikolaos/vtracker.html

```
CONSTRUCT{?release code:hasNonBreakingChangeCount ?totalNonBreakingChanges
}
WHERE{
{
  SELECT ?release count(?nonbreakingChange) as ?totalNonBreakingChanges
  WHERE{
    ?nonbreakingChange rdf:type code:NonBreakingChange.

    ?nonbreakingChange code:hasCurrentAPI ?api.
    ?release code:containsCodeEntity ?api.
  }GROUP BY ?release
}}
```

Listing 23: SPARQL query for inferring the total number of non-breaking changes in a project.

BCI$_{direct}$ and BCI$_{indirect}$ inference: The queries in Listings 24 and 25 take advantage of the inference services to deriv both the direct and indirect BCI scores from a project and its dependencies. The query in Listing 22 first identifies two unique releases of the same project for which breaking changes have been populated into the triple-store. It then identifies any usage of the found binary incompatible APIs within the client. These queries are based on Equations 7 and 8 in Section 7.3.3.

```
CONSTRUCT{?release code:hasDirectBCI ?directBCI }
WHERE{
{

  SELECT ?release ?directBCI

  WHERE {

    BIND((?usedBreakingChanges/?bcc) AS ?directBCI).

    {
      SELECT ?release count(?breakingApi) as ?usedBreakingChanges ?bcc
      WHERE{
        ?breakingChange rdf:type code:BreakingChange.

        ?breakingChange code:hasCurrentAPI ?breakingApi.
        ?dependent code:containsCodeEntity ?breakingApi.
        ?dependent code:hasBreakingChangeCount ?bcc.
        ?client code:containsCodeEntity ?api.
        ?api main:dependsOn ?breakingApi.
      }GROUP BY ?release

    }

  }
}}
```

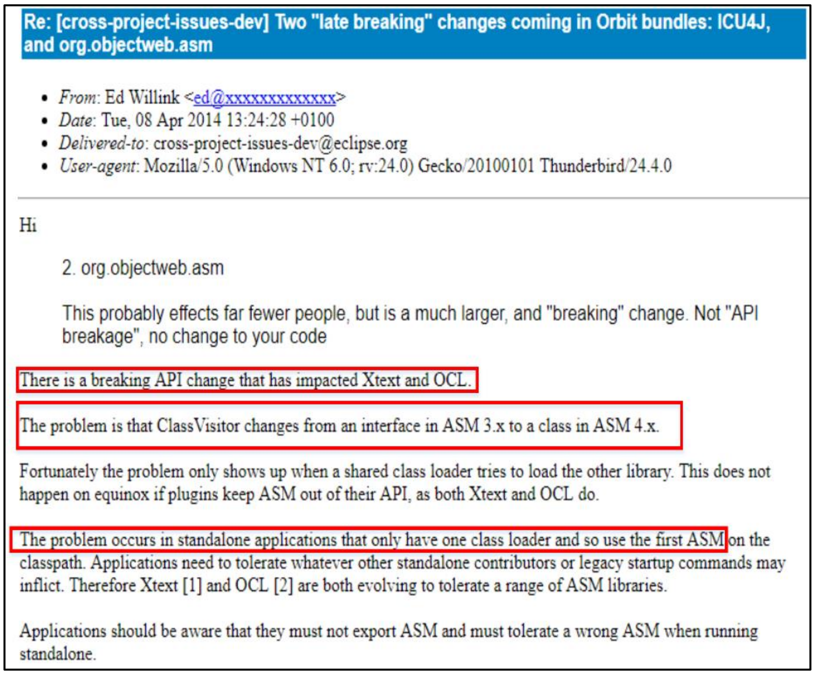
Listing 24: SPARQL query for inferring the BCIdirect measure in a project.

```
CONSTRUCT{?client  code:hasIndirectBCI ?indirectBCI }
WHERE{
{

  SELECT ?client ?indirectBCI

  WHERE {

    BIND((?usedBreakingChanges/?bcc) AS ?indirectBCI).

    {
      SELECT ?client count(?clientAPIEntity) as ?usedBreakingChanges count(?breakingChange) as ?bcc
      WHERE{
        #identify use of breaking change entity in clien
        ?client code:containsCodeEntity ?clientAPIEntity.
        {?clientAPIEntity main:dependsOn ?currentAPIElement} UNION
        {?clientAPIEntity main:dependsOn ?priorAPIElement}.
        {
          SELECT ?client, ?dependency ?asm1, ?asm2
          WHERE {
            #Identify different releases of the same project for which breaking changes exist

            ?client build:hasBuildDependencyOn ?dependency1; build:hasBuildDependencyOn ?dependency2.
            ?breakingChange a code:BreakingCodeChange.
            ?breakingChange code:hasPriorAPI ?priorAPIElement; code:hasCurrentAPI ?currentAPIElement.
            ?dependency1 code:containsCodeEntity ?currentAPIElement.

            ?dependency2 code:containsCodeEntity ?priorAPIElement.
            FILTER(?dependency1 != ?dependency2).
          }
        }
      }

    }

}}
```

Listing 25: SPARQL query for inferring the BCIindirect measure in a project.

Findings and Discussion. Figure 60 shows an example of a bug[93] reported in Eclipse Orbit[94]. Orbit depends on ASM[95], a Java bytecode manipulation library. ASM introduced breaking changes in its later releases, such as ClassVisitor being changed from an interface (version 3.X) to a class in version 4.0. This change is a major change in the API and therefore breaking the older 3.X API releases.

[93] https://dev.eclipse.org/mhonarc/lists/cross-project-issues-dev/msg10487.html
[94] https://www.eclipse.org/orbit/
[95] http://asm.ow2.org/



Re: [cross-project-issues-dev] Two "late breaking" changes coming in Orbit bundles: ICU4J, and org.objectweb.asm

- *From*: Ed Willink <ed@xxxxxxxxxxxxx>
- *Date*: Tue, 08 Apr 2014 13:24:28 +0100
- *Delivered-to*: cross-project-issues-dev@eclipse.org
- *User-agent*: Mozilla/5.0 (Windows NT 6.0; rv:24.0) Gecko/20100101 Thunderbird/24.4.0

Hi

> 2. org.objectweb.asm
>
> This probably effects far fewer people, but is a much larger, and "breaking" change. Not "API breakage", no change to your code

There is a breaking API change that has impacted Xtext and OCL.

The problem is that ClassVisitor changes from an interface in ASM 3.x to a class in ASM 4.x.

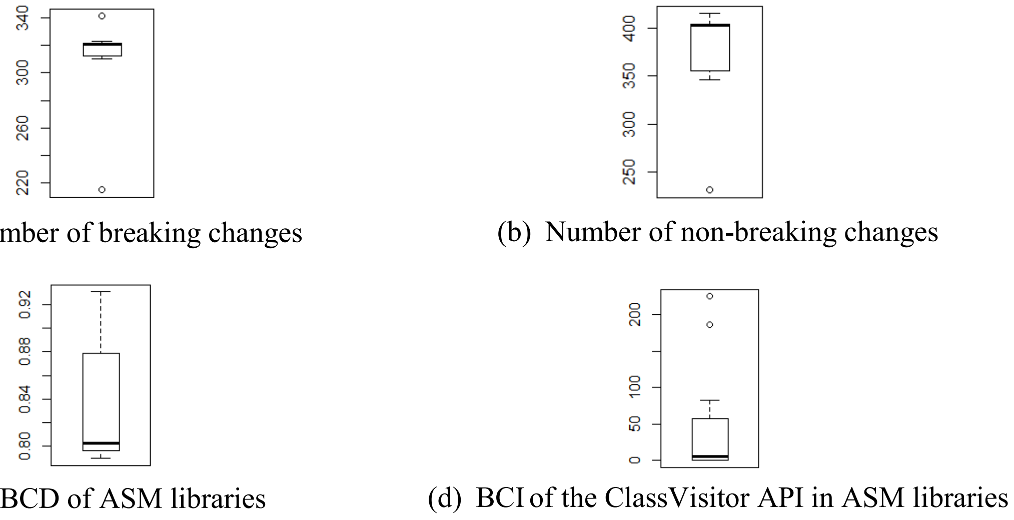
Fortunately the problem only shows up when a shared class loader tries to load the other library. This does not happen on equinox if plugins keep ASM out of their API, as both Xtext and OCL do.

The problem occurs in standalone applications that only have one class loader and so use the first ASM on the classpath. Applications need to tolerate whatever other standalone contributors or legacy startup commands may inflict. Therefore Xtext [1] and OCL [2] are both evolving to tolerate a range of ASM libraries.

Applications should be aware that they must not export ASM and must tolerate a wrong ASM when running standalone.

Figure 60: An example of a reported bug showing how a breaking change in the ASM library impacts Orbit and its dependent projects.

We illustrate how our ontology-based API dependency measures can aid developers in detecting and dealing with such breaking changes. For the analysis, we extract and populate facts about the breaking changes between different versions of ASM releases and the source code of all projects which depend on ASM releases (81,09 dependencies in total). Based on the extracted source code and dependency information, the earlier introduced SPARQL queries can now be used to identify the potential direct and indirect impacts of ASM breaking changes on client applications.



(a) Number of breaking changes

(b) Number of non-breaking changes

(c) BCD of ASM libraries

(d) BCI of the ClassVisitor API in ASM libraries

Figure 61: Distribution of breaking changes and their impacts in the analyzed ASM libraries.

Figure 61 shows the distribution of (a) breaking changes, (b) non- breaking changes, and (c) breaking change densities (BCD) across all selected 20 ASM releases. Figure 61(d) reports on the impact of the ClassVisitor API breaking change on client applications. Furthermore, this particular change can potentially affect on average 50 different API elements, and as many as 225 elements in a single client application. The reported impact set returned by our approach would include clients which reuse the ClassVisitor API either directly (through an implementation of the interface) or indirectly (through transitive inheritance or method invocations).

7.4.5 Assessment Process

The above sub-sections described how we can identify and measure different attributes of trustworthiness by taking advantage of our unified ontological knowledge representation and SW reasoning services. The OntTAM assessment process further integrates these scores across attributes and sub-factors. For the actual assessment process, we first compute the fuzzy score for each measure individually and then aggregate these scores to calculate the attribute, sub-factors, factors, and dimension assessment scores. Figure 62 below gives a complete overview of how the sub-factors, attributes, and measures are related and used to derive our trustworthiness assessment. It should be noted that we do not report on actual trustworthiness scores, since these scores would require a particular assessment context and an instantiation of our OntTAM assessment model with more measures, attributes and sub-factors.

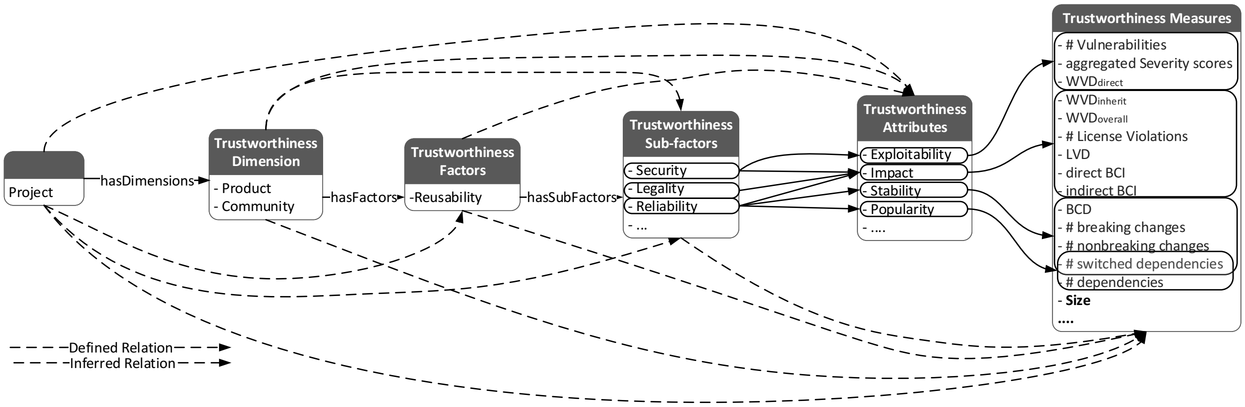


Figure 62: Overview of relations in the semantic OntTAM domain model.

The effect of the fuzzification on the assessment scores typically increases with assessment abstraction levels (e.g., quality dimension scores vs attribute scores). Listing 26 shows the rules we used to create the fuzzified score for the WVD measure, and Listing 27 provides example rules we used to combine the fuzzified LVC and WVD scores into a score for the *Impact* attribute.

```
FUNCTION_BLOCK WVD
VAR_INPUT
  WVD_Measure: REAL;
  WVD_Weight: REAL;
END_VAR
VAR_OUTPUT
  WVD_Score: REAL;
END_VAR
FUZZIFY WVD_Measure
  TERM VERYLOW := (0.0,1.0) (1.04,1.0) (2.11,0.0) ;
  TERM LOW := (1.90,0.0) (2.975,1.0) (4.14,0.0) ;
  TERM AVERAGE := (3.73,0.0) (4.91,1.0) (6.17,0.0) ;
  TERM HIGH := (5.55,0.0) (6.845,1.0) (8.20,0.0) ;
  TERM VERYHIGH := (7.38,0.0) (8.78,1.0) (11.29,1.0)
;
END_FUZZIFY
FUZZIFY WVD_Weight
  TERM LOW := (0.0,1.0) (0.5,1.0) (2.69,0.0) ;
  TERM MEDIUM := (2.56,0.0) (4.75,1.0) (7.05,0.0) ;
  TERM HIGH := (6.69,0.0) (9.0,1.0) (12.0,1.0) ;
```

```
RULEBLOCK WVD_SCORE_RULES
RULE 0 : IF WVD_Measure IS VERYLOW AND
WVD_Weight IS LOW      THEN   WVD_Score IS
EXCELLENT ;
RULE 1 : IF WVD_Measure IS VERYLOW AND
WVD_Weight IS MEDIUM THEN WVD_Score IS
EXCELLENT ;
RULE 2 : IF WVD_Measure IS VERYLOW AND
WVD_Weight IS HIGH THEN WVD_Score IS VERYGOOD ;
RULE 3 : IF WVD_Measure  IS LOW AND WVD_Weight IS
LOW THEN WVD_Score IS EXCELLENT ;
RULE 4 : IF WVD_Measure  IS LOW AND WVD_Weight IS
MEDIUM THEN WVD_Score IS VERYGOOD ;

RULE 5 : IF WVD_Measure  IS LOW AND WVD_Weight IS
HIGH THEN WVD_Score IS AVERAGE ;
RULE 6 : IF WVD_Measure  IS AVERAGE AND
WVD_Weight IS LOW THEN WVD_Score IS VERYGOOD ;
RULE 7 : IF WVD_Measure  IS AVERAGE AND
WVD_Weight IS MEDIUM THEN WVD_Score IS AVERAGE
```

```
END_FUZZIFY
DEFUZZIFY WVD_Score
  TERM VERYPOOR := (6.5,0.0) (7.5,1.0) (9.0,1.0) ;
  TERM POOR := (5.31,0.0) (6.25,1.0) (7.22,0.0) ;
  TERM AVERAGE := (4.14,0.0) (5.0,1.0) (5.9,0.0) ;
  TERM VERYGOOD := (2.95,0.0) (3.75,1.0) (4.6,0.0) ;
  TERM EXCELLENT := (0.0,1.0) (2.5,1.0) (3.28,0.0) ;
  METHOD : COG;
END_DEFUZZIFY
```

```
;
RULE 8 : IF WVD_Measure  IS AVERAGE AND
WVD_Weight IS HIGH THEN WVD_Score IS POOR;
RULE 9 : IF WVD_Measure  IS HIGH AND WVD_Weight IS
LOW THEN WVD_Score IS AVERAGE ;
RULE 10 : IF WVD_Measure  IS HIGH AND WVD_Weight IS
MEDIUM THEN WVD_Score IS POOR ;
RULE 11 : IF WVD_Measure  IS HIGH AND WVD_Weight IS
HIGH THEN WVD_Score IS  VERYPOOR;
RULE 12 : IF WVD_Measure  IS VERYHIGH AND
WVD_Weight IS LOW THEN WVD_Score IS POOR ;
RULE 13 : IF WVD_Measure  IS VERYHIGH AND
WVD_Weight IS MEDIUM THEN WVD_Score IS
VERYPOOR ;
RULE 14 : IF WVD_Measure  IS VERYHIGH AND
WVD_Weight IS HIGH THEN WVD_Score IS VERYPOOR ;
END_RULEBLOCK
END_FUNCTION_BLOCK
```

Listing 26: Sample FCL file for creating fuzzy scores for the WVD measure.

```
RULEBLOCK IMPACT _SCORE_RULES
RULE 0 : IF LVC_Score IS EXCELLENT AND WVD_Score IS VERYPOOR
THEN   IMPACT_Score IS AVERAGE ;
RULE 1 : IF LVC_Score IS VERYGOOD AND WVD_Score IS VERYPOOR
THEN   IMPACT_Score IS POOR ;
RULE 2 : IF LVC_Score IS AVERAGE AND WVD_Score IS VERYPOOR THEN
IMPACT_Score IS POOR ;
RULE 3 : IF LVC_Score IS POOR AND WVD_Score IS VERYPOOR THEN
IMPACT_Score IS VERYPOOR ;
RULE 4 : IF LVC_Score IS VERYPOOR AND WVD_Score IS VERYPOOR
THEN   IMPACT_Score IS VERYPOOR;
…
END_RULEBLOCK
END_FUNCTION_BLOCK
```

Listing 27: Sample FCL file for integrating the LVC and WVD fuzzy scores for the Impact attribute.

Using the property chain axioms explained in Section 7.3.2.1, one can now automatically infer trustworthiness scores from the populated measures of any given project. Listing 28 provides a list of sample queries used for integration and fuzzification.

```
Query 1: At attribute level
SELECT distinct ?project ?impactScore
WHERE {
  ?impactAttribute a onttam:Impact.
  ?project onttam:hasAttribute ?impactAttribute.
  ?impactAttribute onttam:hasScore  ?impactScore.
  FILTER (?impactScore = “EXCELLENT”).
}

Query 2: At factor level
SELECT distinct ?project ?factorScore
WHERE {
  ?factorAttribute a onttam:Factor.
  ?project onttam:hasFactor ?factorAttribute.
  ?factorAttribute onttam:hasScore  ?factorScore.
  FILTER (?factorScore = “EXCELLENT”).
}
```

Listing 28: SPARQL query illustrating the inference of overall trustworthiness scores.

7.5 Chapter Summary

In summary, we introduced OntTAM, a trustworthiness assessment model which is an instantiation of our SE-EQUAM assessment model. OntTAM takes advantage of our SV-AF, a unified knowledge representation of different SE knowledge resources and SVDBs, and extends these knowledge bases to allow for an automated analysis and assessment of trustworthiness quality attributes. We further presented a concrete instantiation of our assessment model that not only provides a formal modeling of trustworthy quality attributes but can also be extended/customized to specific stakeholder needs. We illustrated how a concrete instantiation of OntTAM for a small subset of sub-factors, attributes, and measures related to the trustworthiness of reusable components can be created. The measures which we included in the study are: API breaking changes, security vulnerabilities, and license violations.

In the next chapter, we conclude the thesis and discuss some possible future works.

Chapter 8

8 Conclusions and Future Work

In this chapter, we summarize the findings of this research and discuss some promising directions for future work.

8.1 Summary of the Findings

- We conducted a comprehensive review of the SE literature which focused on the question: “to what extent do SE researchers use the vulnerabilities information hosted in public SVDBs in their research”. From our survey we observed that:
    - There is an increasing awareness of SVDBs in the research community in terms of papers being published describing the use and application of SVDBs in the SE domain.
    - The majority of surveyed articles (91%) used *common* SVDBs in their work, whereas only 9% relied on *specialized* SVDBs.
    - Common SE repositories used in combination with SVDBs are source code and bug repositories.
    - Most of the surveyed studies applied SVDBs only to a limited number of SE activities. We found the most popular SE tasks in the surveyed articles were empirical research (37% of articles), modeling (20% of articles), source code analysis (for static/dynamic vulnerability analysis 16% of articles), and testing (14% of articles).

    - Most studies relied on only one SVDB for their contribution. The common use of SVDBs in these SE tasks was extracting vulnerability examples for validating the assumptions proposed by authors and comprehending the security vulnerability affecting the software system. Also, studies on vulnerability repositories focused on harvesting statistical trends or creating vulnerability models and using them for prediction. Other studies focused on the vulnerability reporters who possess the most important information.

- We introduced a novel knowledge engineering methodology using Formal Concept Analysis (FCA) to semi-automate the software vulnerabilities knowledge acquisition and extraction from SVDBs.
    - We conducted a literature survey of existing software security vulnerability ontologies.
        - Our comparison shows that only 50% of the reviewed vulnerabilities ontologies are publicly available online.
        - None of the surveyed ontologies specified that they used a systematic knowledge engineering approach while developing their ontologies. Most of the presented ontologies are based on the author's experience in the vulnerability domain.
        - Our survey also shows that most papers refer to public advisories (e.g., SVDBs) as their main source of vulnerability information. However, none of the papers explained in detail how the SVDB(s) was used in their knowledge engineering methodology.
        - Many articles only referred indirectly to SVDBs while describing their general background section and did not include an actual use of the SVDBs.
    - We proposed SEVONT, an abstraction hierarchy of software security vulnerabilities analysis ontologies. We proposed a semi-automated methodology using FCA to create a unified ontological knowledge model (SEVONT) that supports knowledge sharing, linking, and inference across SVDB boundaries.
    - We presented alignment rules to facilitate knowledge integration and improve our overall knowledge design.

    - We illustrated the applicability of our modeling approach by providing examples of how our modeling approach supports vulnerability analysis across individual SVDBs.
        - Use case #1: Showed the benefit of using our knowledge modeling approach to link different SVDBs and investigate the vulnerability disclosure date issue. The study shows that 21,654 CVEs appear in two different SVDBs (D1 and D2) and that of these, 16,337 (75%) are disclosed in D2 prior to being published in D1 (with a median number of seven days between the two reports). On the other hand, 3,848 (18%) of vulnerabilities were published first in D1 before they were included in D2. Only 7% of the vulnerabilities show the same publication date in both SVDBs. Our results show that users who rely only on D1 and are unaware of exploits reported in D2 will not receive any alerts from D1.
        - Use case #2: Is an extension of use case #1, which showed that our approach can automatically infer missing vulnerability information in D2 such as the exploit type (described by CWE standard) and the exploit severity score (described by CVSS standard).

- We developed a Security Vulnerability Analysis Framework (SV-AF) to support evidence-based vulnerability detection. The framework has the following achievements:
    - Integrate different knowledge sources ontologies such as build systems ontologies, source code ontologies, version systems ontologies, etc.
    - Implement ontologies alignment using Probabilistic Soft Logic (PSL) framework which establishes weighted links between ontologies.
    - Evaluated with two case studies to illustrate the applicability of the presented approach. We identified that 750 Maven project releases are directly affected by known security vulnerabilities and by considering transitive dependencies, an additional 415,604 Maven projects can be identified as potentially affected by these vulnerabilities.
- We introduced a novel approach for automatically tracing source code vulnerabilities at the API level across project boundaries.

  - We extended our previous SV-AF framework with knowledge from other repositories, such as version control systems (VCS) and build systems to provide additional analysis services such as: (1) identifying and tracing the use of vulnerable code in APIs to projects, and (2) providing notifications about vulnerabilities found in APIs (and their dependent component) that can affect a specific project.
- We introduced a novel Ontological Trustworthiness Assessment Model (OntTAM) which is integrated with SV-AF to 1) support the automated analysis and assessment of quality attributes related to the trustworthiness of libraries and APIs in open-source systems, and 2) provide developers with additional insights into the potential impact of reused libraries and APIs on the quality and trustworthiness of their projects. We illustrate the applicability of our approach by assessing the trustworthiness of libraries in terms of their API breaking changes, security vulnerabilities, license violations, and their potential impact on client projects.

8.2 Future Work

8.2.1 Current Limitations

Users studies. While we performed several case studies to show the applicability of the proposed knowledge model, the fact that we did not incorporate developers' opinions in our designed model and the lack of conducted studies remains a limitation of our work. For example, a controlled user-study should be performed to evaluate how easy it is to integrate our knowledge model into current software development processes. In addition, a user study involving developers of open source projects should be performed to evaluate the usefulness of our vulnerability dependency analysis in improving the trustworthiness of these systems.

The presented approach for identifying transitive dependencies in Chapters 5, 6, and 7 might not be generalizable for non-Maven projects since the case studies we conducted were limited to the use of the Maven dependency management system. However, given the flexibility and openness of our knowledge modeling approach, dependency information from build repositories or resources other than Maven can also be integrated into our approach. The quality of our analysis will however depend on the ability to extract these dependencies accurately. While the fact extraction process for other build systems (e.g., Ant[96], Gradle[97], and MSBuild[98]) differs from the one we used for Maven, the core domain concepts remain the same for these repositories.

Another threat to the validity of 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.

Design Quality. One of the major benefits of our unified vulnerability metamodel approach SEVONT (Chapter 4) is its underlying formalism (machine-readable) and the resulting ease of reuse. While machine readable, the reuse of ontologies can be only partially automated and still requires an ontology expert in order to extend and validate the new ontology design. In particular, modeling new constraints and relations or support for inferring knowledge that is not explicitly modeled in the metamodel requires expertise in ontology modeling and reasoning. We believe that this threat is not unique to our domain and can be observed in other modeling domains (e.g., software design, database design), where the quality of the final model/design similarly depends mostly on the expert performing the design/modeling step. However, we partially mitigate this threat by using FCA theory for our SEVONT's system and domain ontologies integration, and predefined semantic rules for knowledge integration (discussed in Chapter 4).

Another major benefit of our approach is its ability to seamlessly integrate and reuse ontologies while maintaining the quality of the resulting knowledge model. While assessing the quality of an ontology design, or even any design in general, is an inherently difficult problem since what constitutes a quality design will depend on different non-functional requirements (e.g., reuse, usability, extensibility, expressiveness, and reasoning support). We partly addressed this threat by using existing reasoners (such as Pellet, Hermit, and JFact) and tools (OOPS![99] and the Neon Toolkit[100]) to check our ontology design for taxonomic, syntactical, and consistency

[96] https://ant.apache.org/

[97] https://gradle.org/

[98] https://docs.microsoft.com/en-ca/cpp/build/msbuild-visual-cpp

[99] http://oops.linkeddata.es/advanced.jsp

[100] http://neon-toolkit.org/wiki/Download/2.5.2.html

problems. To determine if our ontology constraints were sufficient to identify incorrect data, we incrementally populated the ontologies with facts during the evaluation process. While the reasoners did not report any inconsistencies in our ontologies, OOPS! reported a few problems in our ontologies referring to some violations of design rules found in the OOPS! rule catalogue. However, these identified violations were only related to some missing license information and annotations (such as rdfs:label and rdfs:comment) for some of our ontology elements.

Another potential threat to our research is whether the set of concepts we considered in SV-AF is sufficient to capture the semantics of the analyzed domains. There is always a trade-off in the design of knowledge bases in terms of their expressivity and their usefulness; an equilibrium should be established between the amount of information that is sufficient to accomplish a task and the granularity of the knowledge that should be available to produce useful results. We addressed this threat by showing that our modeled concepts are sufficient to provide flexible analysis services through the described case study experiments.

Validating the correctness of the newly inserted knowledge (e.g., Chapter 7), such as adding a vulnerable project that is not actually a vulnerable project is yet another potential threat. This threat can only be partially mitigated by adding rules and constraints against the populated concepts, since much of the interpretation of what constitutes a vulnerable project in an assessment model is subjective to human interpretation and the specific assessment context.

A potential threat to the trustworthiness approach (Chapter 7) is whether the set of measures we considered in our assessment as part of OntTAM evaluation is sufficient to capture trustworthiness as a factor. We addressed this threat by selecting our trustworthiness measures from a well-established subset of existing trustworthiness models, such as PAS 754:2014, QualiPSo [261], and Boland et. al. [262]. While we only selected a very small subset of these trustworthiness attributes, we believe this subset is sufficient to illustrate the applicability of our assessment model. The objective of our study was not to verify the assessment model for its completeness but rather to demonstrate that OntTAM can be instantiated to a given (user specific) assessment context. The study also shows that instantiating and extending OntTAM to support other requirements including new measures, attributes, or sub-factors is a straightforward task.

Mining SVDBs and SE Repositories. The research presented in this thesis relies on the ability to mine facts from the vulnerabilities databases and SE repositories to populate our ontologies. A common problem with mining software repository is that these repositories often contain noise in their data due to ambiguities, inconsistencies, or incompleteness. For our studies, we were able to mitigate this threat since vulnerabilities published in SVDBs (e.g., NVD) are manually validated and managed by security experts. Also the SE data we used in Chapter 5, 6, and 7 was based on the Maven repository that captures dependencies related to a particular build file. A key premise of Maven is that its dependencies are fully specified and available, therefore eliminating ambiguities and inconsistency in the dataset.

Other threats to the mining of these repositories are related to the fact that we only extracted vulnerabilities reported from 2002 to 2017 from the SVDBs (e.g., NVD) datasets. While our selected data range may not be generalizable for all vulnerabilities, it does cover the majority of all published vulnerabilities. Furthermore, the dataset also covers a broad range of projects and vulnerabilities, ensuring that the dataset can be considered large enough to avoid any bias towards certain vulnerabilities or affected libraries.

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

The case studies introduced in this research are limited in their scope to open source Java projects, and the results obtained from these studies might therefore not be generalizable to other programming languages or system types. Given that our modeling approach is based on different levels of abstraction, we also abstract common aspects of source code and build dependencies in our knowledge model. Currently, we do model object-oriented programming languages, software vulnerabilities, software licenses, and build repositories at the domain-specific layer of our knowledge model. In our modeling approach, these ontologies can be easily extended to specific system level ontologies to model and capture knowledge specific to any object-oriented language, build repository, or vulnerability database.

8.2.2 Opportunities for Future Research

The presented research involves different areas of computer science, including SW technologies, knowledge model, mining software repositories, and source code analysis. This diversity of topics also leads to multiple research directions in which the work presented in this thesis can be extended as part of future work.

8.2.2.1 An Ontology-based Approach to Automate Tagging of Software Artifacts

SE repositories contain a wealth of textual information such as source code comments, developers' discussions, commit messages, and bug reports. These free form text descriptions can contain both direct and implicit references to security concerns. The goal is to derive an approach to extract security concerns from textual information that can yield several benefits, such as bug management (e.g., prioritization), bug triage, or capturing zero-day attack. As part of our ongoing research we have already proposed an automated tagging approach which relies on a semantic ground for the tagging process. More specifically, we introduce a methodical approach for automatically classifying and tagging security concerns found in free-form SE artifacts (e.g., bug reports) using an optimized topic model algorithm [263]. Our approach relies on a variation of the original Latent Dirichlet Allocation (LDA) [264] machine learning algorithm, the Seeded-LDA[263]. Seeded-LDA improves topic detection accuracy by incorporating previous information, by using seeds (known set of concepts) that bear a positive or negative polarity for a given topic domain. These seeds can be obtained from various resources, such as paradigm word lists, a full subjectivity lexicon, or filtered subjective lexica. In our case, we take advantage of a specialized lexicon of security terms provided by the Common Weakness Enumeration (CWE[101]) dictionary. This dictionary describes publicly known information related to security vulnerabilities and exposures. The main contribution of our approach is that it can extract such

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

security concerns from free-form text descriptions without requiring any supervised training data. More specifically, we map all words in a software document (e.g., bug reports), not just entities, to a set of ontological concepts. Using the word-concept distributions we can now apply an entirely unsupervised labeling approach to our dataset. Our preliminary experiments involving the tagging of bug reports show that our approach can extract relevant security concepts, thus reducing the manual effort required in classifying bug reports while at the same tagging them automatically with more representative and standardized security tags. As part of these studies, we mapped these extracted security concepts (cybersecurity ontology) to SV-AF (Chapter 5). Given this unified ontology representation, we can now take advantage of SW [13] reasoners to further infer both implicit and explicit semantic links between the extracted cybersecurity information and other software artifacts. Furthermore, the ontological representation also allows for both extensibility and reuse of our knowledge model for different application contexts.

Note: An earlier version of the work completed in this section has been published in the 11th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM 2017) [265].

As part of our future work, the attack pattern ontology can be developed and combined with cybersecurity tagging ontology and integrated with SV-AF. This will lead to a more powerful vulnerability analysis model; we plan to investigate potential vulnerability patterns based on the usage of vulnerable components. These patterns will provide us with additional insights into assessing and predicting the quality of software systems.

8.2.2.2 SE-GPS: Semantic Global Problems Scanner Visualization Tool

One future research direction is developing a tool that is able to visualize both direct and indirect vulnerable dependencies. In particular, this future work proposes a tool to better understand how much security vulnerabilities affect APIs directly, with the aim of providing more insights into how such API dependencies may be affected by security vulnerability indirectly. We partially implemented some of the suggestions provided by our study from Chapter 3. Our proposed visualization tool provides two visualizations addressing two viewpoints, specifically:

1. Direct vulnerabilities visualization, which helps to identify the security vulnerabilities that affect the specified APIs and shows the severity levels along with their disclosure dates.
2. Transitive vulnerable dependencies visualization, which provides some insights about how the analyzed API depends on other vulnerable APIs components and shows the security vulnerabilities attached to these components.

We implemented SE-GPS, a web-based tool available online [215], that integrates with our SV-AF framework introduced earlier in this thesis. The tool focuses on the visualization vulnerabilities (published by NVD) in the form of direct and indirect affected components.

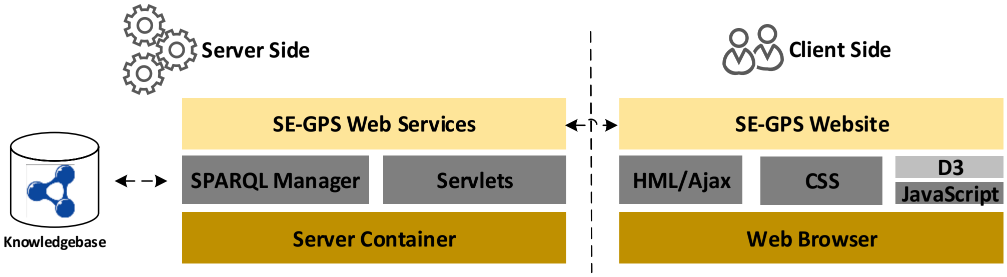


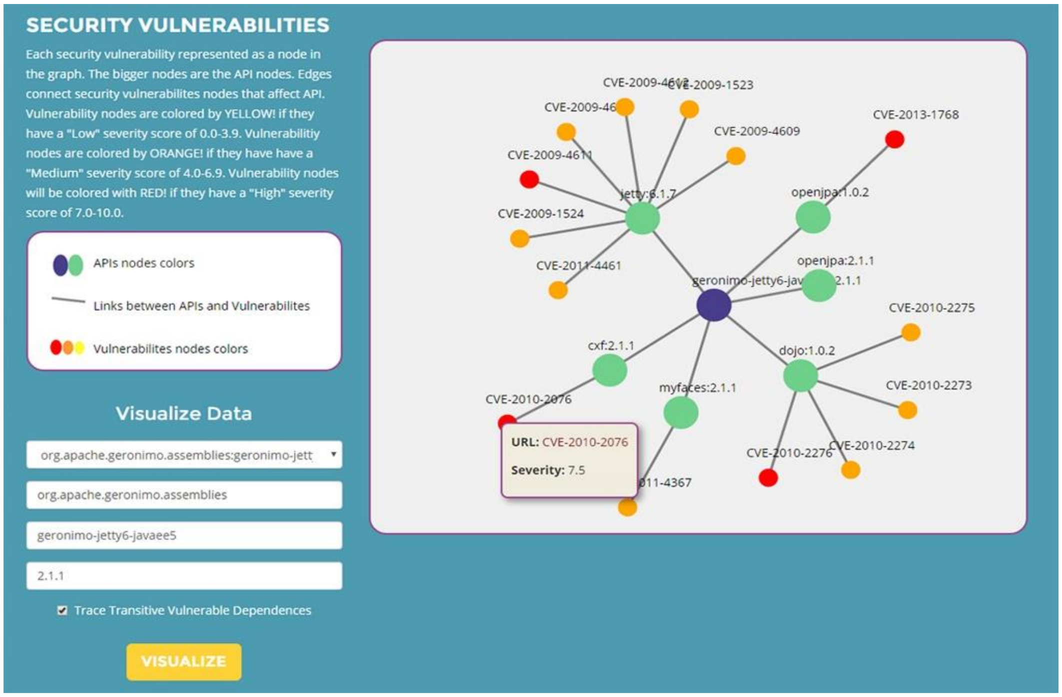
Figure 63: Tool Architecture.

Figure 63 depicts the main components of SE-GPS architecture. As can be seen, SE-GPs is decomposed into a server side, which includes the knowledge base SPARQL[102] engine and the web services, and a client side with the website. Figures 64 and 65 show the current visualizations that SE-GPS offers.

Figure 64 shows the transitive vulnerable dependencies for a given project (e.g., Geronimo-jetty6-javaee5 version 2.1.1). In fact, Geronimo-jetty6-javaee5 version 2.1.1 itself has no reported vulnerability but some of the external APIs contain vulnerabilities that might affect the project.

Figure 65 shows the vulnerabilities associated with a component (in this example Tomcat version 7.0.42) and the node color indicating the number of vulnerabilities affecting a node (e.g., dark color corresponding to a large number of vulnerabilities; in this case Tomcat 7.0.42 is affected by 10 security vulnerabilities). In future work we plan to further extend SE-GPS by creating an Eclipse as a plugin which will provide developers contextual vulnerability notifications within their development process.

[102] Short definition for SPARQL



SECURITY VULNERABILITIES

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

APIs nodes colors

Links between APIs and Vulnerabilites

Vulnerabilites nodes colors

Visualize Data

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

org.apache.geronimo.assemblies

geronimo-jetty6-javaee5

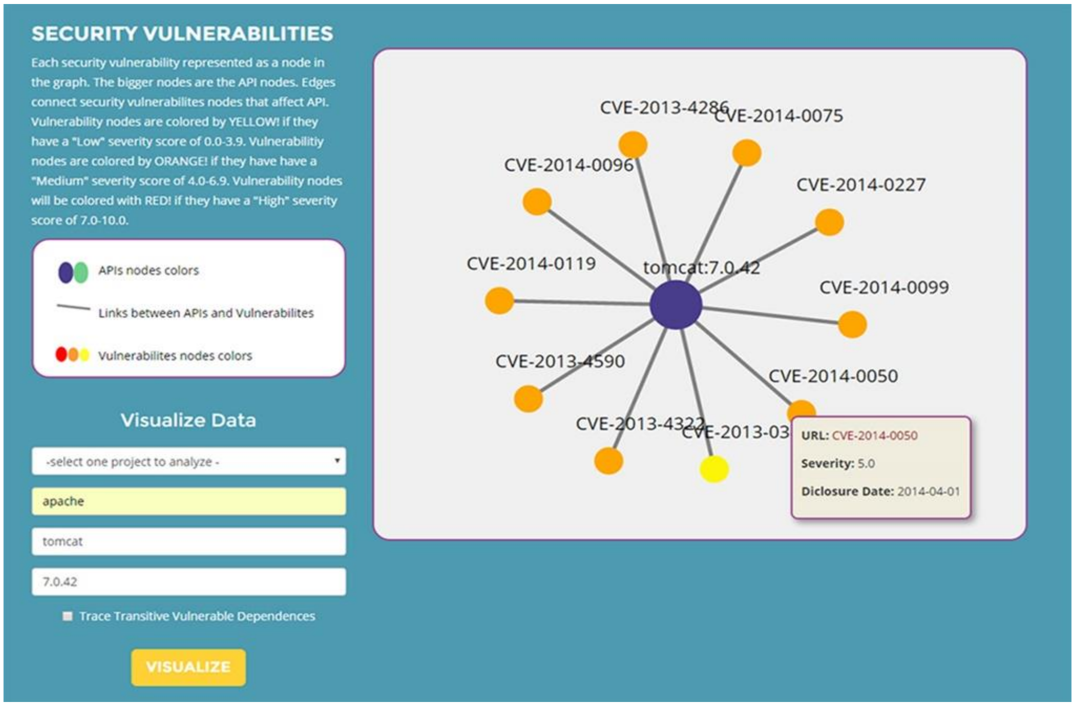
2.1.1

☑ Trace Transitive Vulnerable Dependences

URL: CVE-2010-2076

Severity: 7.5

Figure 64: Visualizing indirect vulnerable dependencies.



SECURITY VULNERABILITIES

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

APIs nodes colors

Links between APIs and Vulnerabilites

Vulnerabilites nodes colors

Visualize Data

-select one project to analyze - ▾

apache

tomcat

7.0.42

□ Trace Transitive Vulnerable Dependences

URL: CVE-2014-0050

Severity: 5.0

Diclosure Date: 2014-04-01

Figure 65: Direct vulnerable dependencies.

Bibliography

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

[2] C. Jones, “Globalization of software supply and demand,” *IEEE Softw.*, pp. 17--24, 1994.

[3] E. and A. Dolstra, “NixOS: A purely functional Linux distribution,” *ACM Sigplan Not.*, vol. 43, pp. 367--378, 2008.

[4] 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.

[5] I. Gutzmer, “Equifax Announces Cybersecurity Incident Involving Consumer Information,” 2017. [Online]. Available: https://investor.equifax.com/news-and-events/news/2017/09-07-2017-213000628. [Accessed: 01-Jun-2018].

[6] A. Goldestein, “The Equifax Breach: Who’s to Blame?,” 2017. [Online]. Available: https://resources.whitesourcesoftware.com/blog-whitesource/the-equifax-breach-who-s-to-blame. [Accessed: 01-Jun-2018].

[7] Snyk, “The State of Open Source Security,” 2017. [Online]. Available: https://snyk.io/stateofossecurity/. [Accessed: 01-Jun-2018].

[8] Sonatype, “2018 DevSecOps Community Survey,” 2018. [Online]. Available: https://www.sonatype.com/2018survey. [Accessed: 01-Jun-2018].

[9] M. Korolov, “Open source software security challenges persist,” 2018. [Online]. Available: https://www.csoonline.com/article/3157377/application-development/open-source-software-security-challenges-persist.html. [Accessed: 01-Jun-2018].

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

[11] R. Laurini, “Pre-consensus Ontologies and Urban Databases,” 2007, pp. 27–36.

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

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

[14] W. O. W. Group, “OWL 2 Web Ontology Language Document Overview (Second Edition),” 2012. [Online]. Available: http://www.w3.org/TR/owl2-overview/. [Accessed: 01-Dec-2014].

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

[16] S. Chabot, “A Review of ‘A Semantic Web Primer,’” *J. Web Librariansh.*, vol. 4, no. 1, pp. 97–98, Mar. 2010.

[17] Apache, “Apache Jena,” 2000. [Online]. Available: https://jena.apache.org/. [Accessed: 10-Jan-2015].

[18] O. Software, “OpenLink,” 1992. [Online]. Available: http://virtuoso.openlinksw.com/. [Accessed: 10-Jan-2015].

[19] J. Aasman, “Allegro graph: RDF triple database,” 2006. [Online]. Available: http://franz.com/agraph/allegrograph/. [Accessed: 10-Jan-2015].

[20] 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.

[21] M. Uschold and M. Gruninger, "Ontologies: principles, methods and applications," *Knowl. Eng. Rev.*, vol. 11, no. 02, p. 93, Jun. 1996.

[22] 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, 2006, pp. 1–48.

[23] F. Ruiz and J. R. Hilera, "Using Ontologies in Software Engineering and Technology," in *Ontologies for Software Engineering and Software Technology*, Springer Berlin Heidelberg, 2006, pp. 49–102.

[24] 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.

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

[26] 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.

[27] 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.

[28] 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.

[29] 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.

[30] 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.

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

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

[33] 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.

[34] B. Ganter and R. Wille, *Formal Concept Analysis*. Berlin, Heidelberg, Heidelberg: Springer Berlin Heidelberg, 1999.

[35] M. Shiri, J. Hassine, and J. Rilling, "A Requirement Level Modification Analysis Support Framework," in *Third International IEEE Workshop on Software Evolvability 2007*, 2007, pp. 67–74.

[36] S. O. Kuznetsov and J. Poelmans, "Knowledge representation and processing with formal concept analysis," *Wiley Interdiscip. Rev. Data Min. Knowl. Discov.*, vol. 3, no. 3, pp. 200–215, May 2013.

[37] J. Vacca, *Computer and Information Security Handbook*. elsevier, 2013.

[38] M. Karlsson, “The Edit History of the National Vulnerability Database and similar Vulnerability Databases,” 2012.

[39] Carlos Vazques, “Auditing Using Vulnerability Tools to Identify Today’s Threats to Business Performance,” 2014.

[40] B. Liu, L. Shi, Z. Cai, and M. Li, “Software Vulnerability Discovery Techniques: A Survey,” in *2012 Fourth International Conference on Multimedia Information Networking and Security*, 2012, pp. 152–156.

[41] T. U. of Maryland., “FindBugs,” 2004. [Online]. Available: http://findbugs.sourceforge.net/. [Accessed: 10-Mar-2015].

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

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

[44] S. S. Alqahtani, E. E. Eghan, and J. Rilling, “SV-AF — A Security Vulnerability Analysis Framework,” in *2016 IEEE 27th International Symposium on Software Reliability Engineering (ISSRE)*, 2016, pp. 219–229.

[45] J. Luszcz, “Apache Struts 2: how technical and development gaps caused the Equifax Breach,” *Netw. Secur.*, vol. 2018, no. 1, pp. 5–8, Jan. 2018.

[46] Schumacher, M. and Haul, C. and Hurler, M. and Buchmann, and Alejandro, “Data Mining in Vulnerability Databases,” *Comput. Sci.*, vol. 12, 2000.

[47] S. S. Alqahtani and J. Rilling, “Survey Dataset,” 2018. [Online]. Available: https://github.com/isultane/Survey-dateset. [Accessed: 10-May-2018].

[48] T.-H. Chen, S. W. Thomas, and A. E. Hassan, “A survey on the use of topic models when mining software repositories,” *Empir. Softw. Eng.*, vol. 21, no. 5, pp. 1843–1919, Oct. 2016.

[49] W. Martin, F. Sarro, Y. Jia, Y. Zhang, and M. Harman, “A Survey of App Store Analysis for Software Engineering,” *IEEE Trans. Softw. Eng.*, vol. 43, no. 9, pp. 817–847, Sep. 2017.

[50] K. Petersen, R. Feldt, S. Mujtaba, and M. Mattsson, “Systematic Mapping Studies in Software Engineering,” in *12th International Conference on Evaluation and Assessment in Software Engineering (EASE)*, 2008, pp. 68--77.

[51] C. Wohlin, “Guidelines for snowballing in systematic literature studies and a replication in software engineering,” in *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering - EASE ’14*, 2014, pp. 1–10.

[52] L. Sampaio and A. Garcia, “Exploring context-sensitive data flow analysis for early vulnerability detection,” *J. Syst. Softw.*, vol. 113, pp. 337–361, Mar. 2016.

[53] N. Palsetia, G. Deepa, F. Ahmed Khan, P. S. Thilagam, and A. R. Pais, “Securing native XML database-driven web applications from XQuery injection vulnerabilities,” *J. Syst. Softw.*, vol. 122, pp. 93–109, Dec. 2016.

[54] J. Bozic, B. Garn, D. E. Simos, and F. Wotawa, “Evaluation of the IPO-Family algorithms for test case generation in web security testing,” in *2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, 2015, pp. 1–10.

[55] J. Walden, J. Stuckman, and R. Scandariato, “Predicting Vulnerable Components:

Software Metrics vs Text Mining," in *2014 IEEE 25th International Symposium on Software Reliability Engineering*, 2014, pp. 23–33.

[56] N. Mendes, H. Madeira, and J. Duraes, "Security Benchmarks for Web Serving Systems," in *2014 IEEE 25th International Symposium on Software Reliability Engineering*, 2014, pp. 1–12.

[57] D. E. Perry, A. A. Porter, and L. G. Votta, "Empirical studies of software engineering," in *Proceedings of the conference on The future of Software engineering - ICSE '00*, 2000, pp. 345–355.

[58] B. J. Berger, K. Sohr, and R. Koschke, "Extracting and Analyzing the Implemented Security Architecture of Business Applications," in *2013 17th European Conference on Software Maintenance and Reengineering*, 2013, pp. 285–294.

[59] J. D. Meier, A. Mackman, and B. Wastell, "Threat Modeling Web Applications," *Microsoft Corporation*, 2005. .

[60] A. Chatzipoulidis, D. Michalopoulos, and I. Mavridis, "Information infrastructure risk prediction through platform vulnerability analysis," *J. Syst. Softw.*, vol. 106, pp. 28–41, Aug. 2015.

[61] R. Scandariato, J. Walden, A. Hovsepyan, and W. Joosen, "Predicting Vulnerable Software Components via Text Mining," *IEEE Trans. Softw. Eng.*, vol. 40, no. 10, pp. 993–1006, Oct. 2014.

[62] L. K. Shar, H. Beng Kuan Tan, and L. C. Briand, "Mining SQL injection and cross site scripting vulnerabilities using hybrid program analysis," in *2013 35th International Conference on Software Engineering (ICSE)*, 2013, pp. 642–651.

[63] 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.

[64] Y. Wu, H. Siy, and R. Gandhi, "Empirical results on the study of software vulnerabilities," in *Proceeding of the 33rd international conference on Software engineering - ICSE '11*, 2011, p. 964.

[65] N. H. Pham, T. T. Nguyen, H. A. Nguyen, X. Wang, A. T. Nguyen, and T. N. Nguyen, "Detecting recurring and similar software vulnerabilities," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - ICSE '10*, 2010, vol. 2, p. 227.

[66] P. Anbalagan and M. Vouk, "Towards a Unifying Approach in Understanding Security Problems," in *2009 20th International Symposium on Software Reliability Engineering*, 2009, pp. 136–145.

[67] H. Cavusoglu, H. Cavusoglu, and S. Raghunathan, "Efficiency of Vulnerability Disclosure Mechanisms to Disseminate Vulnerability Knowledge," *IEEE Trans. Softw. Eng.*, vol. 33, no. 3, pp. 171–185, Mar. 2007.

[68] E. S. Pasaribu, Y. Asnar, and M. M. I. Liem, "Input injection detection in Java code," in *2014 International Conference on Data and Software Engineering (ICODSE)*, 2014, pp. 1–6.

[69] Y. Zheng and X. Zhang, "Path sensitive static analysis of web applications for remote code execution vulnerability detection," in *2013 35th International Conference on Software Engineering (ICSE)*, 2013, pp. 652–661.

[70] A. Møller and M. Schwarz, "Automated Detection of Client-State Manipulation

Vulnerabilities," *ACM Trans. Softw. Eng. Methodol.*, vol. 23, no. 4, pp. 1–30, Sep. 2014.

[71] H. Shahriar and M. Zulkernine, "Client-Side Detection of Cross-Site Request Forgery Attacks," in *2010 IEEE 21st International Symposium on Software Reliability Engineering*, 2010, pp. 358–367.

[72] G. Wassermann and Z. Su, "Static detection of cross-site scripting vulnerabilities," in *Proceedings of the 13th international conference on Software engineering - ICSE '08*, 2008, p. 171.

[73] J. Thome, L. K. Shar, and L. Briand, "Security slicing for auditing XML, XPath, and SQL injection vulnerabilities," in *2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE)*, 2015, pp. 553–564.

[74] R. Wang, P. Liu, L. Zhao, Y. Cheng, and L. Wang, "deExploit: Identifying misuses of input data to diagnose memory-corruption exploits at the binary level," *J. Syst. Softw.*, vol. 124, pp. 153–168, Feb. 2017.

[75] F. Gao, L. Wang, and X. Li, "BovInspector: automatic inspection and repair of buffer overflow vulnerabilities," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering - ASE 2016*, 2016, pp. 786–791.

[76] M. Ceccato, C. D. Nguyen, D. Appelt, and L. C. Briand, "SOFIA: an automated security oracle for black-box testing of SQL-injection vulnerabilities," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering - ASE 2016*, 2016, pp. 167–177.

[77] V.-T. Pham, M. Böhme, and A. Roychoudhury, "Model-based whitebox fuzzing for program binaries," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering - ASE 2016*, 2016, pp. 543–553.

[78] B. Stivalet and E. Fong, "Large Scale Generation of Complex and Faulty PHP Test Cases," in *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, 2016, pp. 409–415.

[79] B. Min and V. Varadharajan, "A New Technique for Counteracting Web Browser Exploits," in *2014 23rd Australian Software Engineering Conference*, 2014, pp. 132–141.

[80] E. Pek and R. Lammel, "A Literature Survey on Empirical Evidence in Software Engineering," *Comput. Res. Repos.*, vol. abs/1304.1, 2013.

[81] M. Hafiz and M. Fang, "Game of detections: how are security vulnerabilities discovered in the wild?," *Empir. Softw. Eng.*, vol. 21, no. 5, pp. 1920–1959, Oct. 2016.

[82] N. Munaiah, F. Camilo, W. Wigham, A. Meneely, and M. Nagappan, "Do bugs foreshadow vulnerabilities? An in-depth study of the chromium project," *Empir. Softw. Eng.*, Aug. 2016.

[83] T. Ye, L. Zhang, L. Wang, and X. Li, "An Empirical Study on Detecting and Fixing Buffer Overflow Bugs," in *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, 2016, pp. 91–101.

[84] M. di Biase, M. Bruntink, and A. Bacchelli, "A Security Perspective on Code Review: The Case of Chromium," in *2016 IEEE 16th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, 2016, pp. 21–30.

[85] M. Jimenez, M. Papadakis, and Y. Le Traon, "Vulnerability Prediction Models: A Case Study on the Linux Kernel," in *2016 IEEE 16th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, 2016, pp. 1–10.

[86] S. S. Murtaza, W. Khreich, A. Hamou-Lhadj, and A. B. Bener, "Mining trends and patterns of software vulnerabilities," *J. Syst. Softw.*, vol. 117, pp. 218–228, Jul. 2016.

[87] F. Camilo, A. Meneely, and M. Nagappan, "Do Bugs Foreshadow Vulnerabilities? A Study of the Chromium Project," in *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*, 2015, pp. 269–279.

[88] M. Fang and M. Hafiz, "Discovering buffer overflow vulnerabilities in the wild," in *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement - ESEM '14*, 2014, pp. 1–10.

[89] L. Tan, C. Liu, Z. Li, X. Wang, Y. Zhou, and C. Zhai, "Bug characteristics in open source software," *Empir. Softw. Eng.*, vol. 19, no. 6, pp. 1665–1705, Dec. 2014.

[90] J. Stuckman and J. Purtilo, "Mining Security Vulnerabilities from Linux Distribution Metadata," in *2014 IEEE International Symposium on Software Reliability Engineering Workshops*, 2014, pp. 323–328.

[91] F. Massacci and V. H. Nguyen, "An Empirical Methodology to Evaluate Vulnerability Discovery Models," *IEEE Trans. Softw. Eng.*, vol. 40, no. 12, pp. 1147–1162, Dec. 2014.

[92] D. Wijayasekara, M. Manic, and M. McQueen, "Vulnerability identification and classification via text mining bug databases," in *IECON 2014 - 40th Annual Conference of the IEEE Industrial Electronics Society*, 2014, pp. 3612–3618.

[93] A. Meneely, H. Srinivasan, A. Musa, A. R. Tejeda, M. Mokary, and B. Spates, "When a Patch Goes Bad: Exploring the Properties of Vulnerability-Contributing Commits," in *2013 ACM / IEEE International Symposium on Empirical Software Engineering and Measurement*, 2013, pp. 65–74.

[94] A. Meneely and S. Lucidi, "Vulnerability of the Day: Concrete demonstrations for software engineering undergraduates," in *2013 35th International Conference on Software Engineering (ICSE)*, 2013, pp. 1154–1157.

[95] D. Y. Lee, M. Vouk, and L. Williams, "Using software reliability models for security assessment - Verification of assumptions," in *2013 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, 2013, pp. 23–24.

[96] M. Shahzad, M. Z. Shafiq, and A. X. Liu, "A large scale exploratory analysis of software vulnerability life cycles," in *2012 34th International Conference on Software Engineering (ICSE)*, 2012, pp. 771–781.

[97] K. Goseva-Popstojanova, G. Anastasovski, and R. Pantev, "Using Multiclass Machine Learning Methods to Classify Malicious Behaviors Aimed at Web Systems," in *2012 IEEE 23rd International Symposium on Software Reliability Engineering*, 2012, pp. 81–90.

[98] Q. Liu, Y. Zhang, Y. Kong, and Q. Wu, "Improving VRSS-based vulnerability prioritization using analytic hierarchy process," *J. Syst. Softw.*, vol. 85, no. 8, pp. 1699–1708, Aug. 2012.

[99] D. Wijayasekara, M. Manic, J. L. Wright, and M. McQueen, "Mining Bug Databases for Unidentified Software Vulnerabilities," in *2012 5th International Conference on Human System Interactions*, 2012, pp. 89–96.

[100] A. Austin and L. Williams, "One Technique is Not Enough: A Comparison of Vulnerability Discovery Techniques," in *2011 International Symposium on Empirical Software Engineering and Measurement*, 2011, pp. 97–106.

[101] B. Smith and L. Williams, "Using SQL Hotspots in a Prioritization Heuristic for Detecting All Types of Web Application Vulnerabilities," in *2011 Fourth IEEE International Conference on Software Testing, Verification and Validation*, 2011, pp. 220–229.

[102] S. Zhang, D. Caragea, and X. Ou, "An Empirical Study on Using the National

Vulnerability Database to Predict Software Vulnerabilities," 2011, pp. 217–231.

[103] S. Zaman, B. Adams, and A. E. Hassan, "Security versus performance bugs," in *Proceeding of the 8th working conference on Mining software repositories - MSR '11*, 2011, p. 93.

[104] T. Huynh and J. Miller, "An empirical investigation into open source web applications' implementation vulnerabilities," *Empir. Softw. Eng.*, vol. 15, no. 5, pp. 556–576, Oct. 2010.

[105] T. Zimmermann, N. Nagappan, and L. Williams, "Searching for a Needle in a Haystack: Predicting Security Vulnerabilities for Windows Vista," in *2010 Third International Conference on Software Testing, Verification and Validation*, 2010, pp. 421–428.

[106] S. Neuhaus and T. Zimmermann, "Security Trend Analysis with CVE Topic Models," in *2010 IEEE 21st International Symposium on Software Reliability Engineering*, 2010, pp. 111–120.

[107] A. Mauczka, C. Schanes, F. Fankhauser, M. Bernhart, and T. Grechenig, "Mining security changes in FreeBSD," in *2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)*, 2010, pp. 90–93.

[108] J. Wal, M. Doyle, G. A. Welch, and M. Whelan, "Security of open source web applications," in *2009 3rd International Symposium on Empirical Software Engineering and Measurement*, 2009, pp. 545–553.

[109] P. Anbalagan and M. Vouk, "On mining data across software repositories," in *2009 6th IEEE International Working Conference on Mining Software Repositories*, 2009, pp. 171–174.

[110] P. Anba and M. Vouk, "An empirical study of security problem reports in Linux distributions," in *2009 3rd International Symposium on Empirical Software Engineering and Measurement*, 2009, pp. 481–484.

[111] G. Vache, "Vulnerability analysis for a quantitative security evaluation," in *2009 3rd International Symposium on Empirical Software Engineering and Measurement*, 2009, pp. 526–534.

[112] R. Telang and S. Wattal, "An Empirical Analysis of the Impact of Software Vulnerability Announcements on Firm Stock Price," *IEEE Trans. Softw. Eng.*, vol. 33, no. 8, pp. 544–557, Aug. 2007.

[113] O. Alhazmi and Y. Malaiya, "Measuring and Enhancing Prediction Capabilities of Vulnerability Discovery Models for Apache and IIS HTTP Servers," in *2006 17th International Symposium on Software Reliability Engineering*, 2006, pp. 343–352.

[114] S. Frei, M. May, U. Fiedler, and B. Plattner, "Large-scale vulnerability analysis," in *Proceedings of the 2006 SIGCOMM workshop on Large-scale attack defense - LSAD '06*, 2006, pp. 131–138.

[115] O. H. Alhazmi and Y. K. Malaiya, "Modeling the Vulnerability Discovery Process," in *16th IEEE International Symposium on Software Reliability Engineering (ISSRE'05)*, 2005, pp. 129–138.

[116] J. D. Musa and K. Okumoto, "A logarithmic poisson execution time model for software reliability measurement," in *ICSE '84 Proceedings of the 7th international conference on Software engineering*, 1984, pp. 230–238.

[117] X.-F. Team, "IBM Internet Security Systems X-Force Threat Insight Quarterly," 2009.

[118] FIRST SIG, "Common Vulnerability Scoring System SIG," 2018. [Online]. Available: https://www.first.org/cvss/. [Accessed: 10-May-2018].

[119] Q. Liu and Y. Zhang, "VRSS: A new system for rating and scoring vulnerabilities," *Comput. Commun.*, vol. 34, no. 3, pp. 264–273, Mar. 2011.

[120] R. France and B. Rumpe, "Model-driven Development of Complex Software: A Research Roadmap," in *Future of Software Engineering (FOSE '07)*, 2007, pp. 37–54.

[121] P. Morrison, "Building a security practices evaluation framework," in *Proceedings of the 2015 Symposium and Bootcamp on the Science of Security - HotSoS '15*, 2015, pp. 1–2.

[122] S. S. Murtaza, A. Hamou-Lhadj, W. Khreich, and M. Couture, "Total ADS: Automated Software Anomaly Detection System," in *2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation*, 2014, pp. 83–88.

[123] A. Milenkoski, B. D. Payne, N. Antunes, M. Vieira, and S. Kounev, "Experience Report: An Analysis of Hypercall Handler Vulnerabilities," in *2014 IEEE 25th International Symposium on Software Reliability Engineering*, 2014, pp. 100–111.

[124] M. Almorsy, J. Grundy, and A. S. Ibrahim, "Automated software architecture security risk analysis using formalized signatures," in *2013 35th International Conference on Software Engineering (ICSE)*, 2013, pp. 662–671.

[125] L. K. Shar and H. B. K. Tan, "Predicting common web application vulnerabilities from input validation and sanitization code patterns," in *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering - ASE 2012*, 2012, p. 310.

[126] M. Almorsy, J. Grundy, and A. S. Ibrahim, "Supporting automated vulnerability analysis using formalized vulnerability signatures," in *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering - ASE 2012*, 2012, p. 100.

[127] F. Gauthier and E. Merlo, "Fast Detection of Access Control Vulnerabilities in PHP Applications," in *2012 19th Working Conference on Reverse Engineering*, 2012, pp. 247–256.

[128] D. Xu and K. E. Nygard, "Threat-driven modeling and verification of secure software using aspect-oriented Petri nets," *IEEE Trans. Softw. Eng.*, vol. 32, no. 4, pp. 265–278, Apr. 2006.

[129] D. Byers, S. Ardi, N. Shahmehri, and C. Duma, "Modeling Software VulnerabilitiesWith Vulnerability Cause Graphs," in *2006 22nd IEEE International Conference on Software Maintenance*, 2006, pp. 411–422.

[130] Y. Wu, R. A. Gandhi, and H. Siy, "Using semantic templates to study vulnerabilities recorded in large software repositories," in *Proceedings of the 2010 ICSE Workshop on Software Engineering for Secure Systems - SESS '10*, 2010, pp. 22–28.

[131] OWASP, "Application Threat Modeling," 2017. [Online]. Available: https://www.owasp.org/index.php/Application_Threat_Modeling. [Accessed: 10-May-2018].

[132] OWASP, "Source Code Analysis Tools," 2018. [Online]. Available: https://www.owasp.org/index.php/Source_Code_Analysis_Tools. [Accessed: 10-May-2018].

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

[134] J. Ming, D. Wu, J. Wang, G. Xiao, and P. Liu, "StraightTaint: decoupled offline symbolic taint analysis," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering - ASE 2016*, 2016, pp. 308–319.

[135] C. Theisen, K. Herzig, P. Morrison, B. Murphy, and L. Williams, "Approximating Attack Surfaces with Stack Traces," in *2015 IEEE/ACM 37th IEEE International Conference on*

Software Engineering, 2015, pp. 199–208.

[136] S. Renatus, C. Bartelheimer, and J. Eichler, “Improving prioritization of software weaknesses using security models with AVUS,” in *2015 IEEE 15th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, 2015, pp. 259–264.

[137] Z. Coker and M. Hafiz, “Program transformations to fix C integers,” in *2013 35th International Conference on Software Engineering (ICSE)*, 2013, pp. 792–801.

[138] E. Ofuonye and J. Miller, “Securing web-clients with instrumented code and dynamic runtime monitoring,” *J. Syst. Softw.*, vol. 86, no. 6, pp. 1689–1711, Jun. 2013.

[139] A. R. Bernat and B. P. Miller, “Structured Binary Editing with a CFG Transformation Algebra,” in *2012 19th Working Conference on Reverse Engineering*, 2012, pp. 9–18.

[140] N. DuPaul, “Static Testing vs. Dynamic Testing,” *veracode*, 2017. [Online]. Available: https://www.veracode.com/blog/2013/12/static-testing-vs-dynamic-testing. [Accessed: 10-May-2018].

[141] R. Auger, “XML Injection,” *Web Application Security Consortium Project*, 2010. [Online]. Available: http://projects.webappsec.org/w/page/13247004/XML Injection. [Accessed: 10-May-2018].

[142] R. Auger, “XPath Injection,” *Web Application Security Consortium Project*, 2010. [Online]. Available: http://projects.webappsec.org/w/page/13247005/XPath Injection. [Accessed: 10-May-2018].

[143] R. Dev, A. Jääskeläinen, and M. Katara, “Model-Based GUI Testing: Case Smartphone Camera and Messaging Development,” 2012, pp. 65–122.

[144] J. Nordholm, “Model-Based Testing: An Evaluation,” 2010.

[145] L. Pesante, “Introduction to Information Security,” *US-CERT*, 2008. [Online]. Available: https://www.us-cert.gov/security-publications/introduction-information-security. [Accessed: 10-May-2018].

[146] D. Appelt, C. D. Nguyen, and L. Briand, “Behind an Application Firewall, Are We Safe from SQL Injection Attacks?,” in *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*, 2015, pp. 1–10.

[147] V.-T. Pham, W. B. Ng, K. Rubinov, and A. Roychoudhury, “Hercules: Reproducing Crashes in Real-World Application Binaries,” in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, 2015, pp. 891–901.

[148] A. Aydin, M. Alkhalaf, and T. Bultan, “Automated Test Generation from Vulnerability Signatures,” in *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation*, 2014, pp. 193–202.

[149] K. Hossen, R. Groz, C. Oriat, and J.-L. Richier, “Automatic Generation of Test Drivers for Model Inference of Web Applications,” in *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops*, 2013, pp. 441–444.

[150] A. Blome, M. Ochoa, K. Li, M. Peroli, and M. T. Dashti, “VERA: A Flexible Model-Based Vulnerability Testing Tool,” in *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*, 2013, pp. 471–478.

[151] F. Lebeau, B. Legeard, F. Peureux, and A. Vernotte, “Model-Based Vulnerability Testing for Web Applications,” in *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops*, 2013, pp. 445–452.

[152] M. Buchler, J. Oudinet, and A. Pretschner, “SPaCiTE -- Web Application Testing Engine,” in *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, 2012, pp. 858–859.

[153] D. Zhang, D. Liu, Y. Lei, D. Kung, C. Csallner, N. Nystrom, and W. Wang, “SimFuzz: Test case similarity directed deep fuzzing,” *J. Syst. Softw.*, vol. 85, no. 1, pp. 102–111, Jan. 2012.

[154] H. Shahriar and M. Zulkernine, “MUTEC: Mutation-based testing of Cross Site Scripting,” in *2009 ICSE Workshop on Software Engineering for Secure Systems*, 2009, pp. 47–53.

[155] A. El-Ahmad and H. Arafeh, “The Influence of Software Risk Management on Software Project Success,” 2017.

[156] 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.

[157] Y. Yu, V. N. L. Franqueira, T. Than Tun, R. J. Wieringa, and B. Nuseibeh, “Automated analysis of security requirements through risk-based argumentation,” *J. Syst. Softw.*, vol. 106, pp. 102–116, Aug. 2015.

[158] J. Cox, E. Bouwers, M. van Eekelen, and J. Visser, “Measuring Dependency Freshness in Software Systems,” in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, 2015, pp. 109–118.

[159] R. Kannavara, “Assessing the Threat Landscape for Software Libraries,” in *2014 IEEE International Symposium on Software Reliability Engineering Workshops*, 2014, pp. 71–76.

[160] R. Kannavara, “Securing Opensource Code via Static Analysis,” in *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, 2012, pp. 429–436.

[161] S. H. Houmb, V. N. L. Franqueira, and E. A. Engum, “Quantifying security risk level from CVSS estimates of frequency and impact,” *J. Syst. Softw.*, vol. 83, no. 9, pp. 1622–1634, Sep. 2010.

[162] C. Fruhwirth and T. Mannisto, “Improving CVSS-based vulnerability prioritization and response with context information,” in *2009 3rd International Symposium on Empirical Software Engineering and Measurement*, 2009, pp. 535–544.

[163] M. Boldt, B. Carlsson, and R. Martinsson, “Software Vulnerability Assessment Version Extraction and Verification,” in *International Conference on Software Engineering Advances (ICSEA 2007)*, 2007, pp. 59–59.

[164] 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.

[165] 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.

[166] S. S. Alqahtani, E. E. Eghan, and J. Rilling, “Recovering Semantic Traceability Links between APIs and Security Vulnerabilities: An Ontological Modeling Approach,” in *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, 2017, pp. 80–91.

[167] S. Neuhaus, T. Zimmermann, C. Holler, and A. Zeller, “Predicting vulnerable software components,” in *Proceedings of the 14th ACM conference on Computer and communications security - CCS ’07*, 2007, p. 529.

[168] S. Neuhaus, T. Zimmermann, and T. Zimmermann, “The Beauty and the Beast:

Vulnerabilities in Red Hat's Packages," in *Proceedings of the 2009 USENIX Annual Technical Conference (USENIX ATC)*, 2009, pp. 383–396.

[169] F. Massacci, S. Neuhaus, and V. H. Nguyen, "After-Life Vulnerabilities: A Study on Firefox Evolution, Its Vulnerabilities, and Fixes," 2011, pp. 195–208.

[170] V. Mulwad, W. Li, A. Joshi, T. Finin, and K. Viswanathan, "Extracting Information about Security Vulnerabilities from Web Text," in *IEEE/WIC/ACM International Conferences on Web Intelligence and Intelligent Agent Technology*, 2011, pp. 257–260.

[171] A. Joshi, R. Lal, T. Finin, and A. Joshi, "Extracting Cybersecurity Related Linked Data from Text," in *IEEE Seventh International Conference on Semantic Computing*, 2013, pp. 252–259.

[172] J. A. Wang and M. Guo, "OVM: an ontology for vulnerability management," in *Proceedings of the 5th Annual Workshop on Cyber Security and Information Intelligence Research Cyber Security and Information Intelligence Challenges and Strategies - CSIIRW '09*, 2009, p. 1.

[173] I. Souag, Amina and Salinesi, Camille and Comyn-Wattiau, A. Souag, C. Salinesi, I. Comyn-Wattiau, and I. Souag, Amina and Salinesi, Camille and Comyn-Wattiau, "Ontologies for Security Requirements: A Literature Survey and Classification," in *Advanced Information Systems Engineering Workshops*, Springer, 2012, pp. 61–69.

[174] C. Blanco, J. Lasheras, R. Valencia-Garc, E. Fern, A. Toval, M. Piattini, and M. Blanco, Carlos and Lasheras, Joaquin and Valencia-Garc{\'\i}a, Rafael and Fern{\'a}ndez-Medina, Eduardo and Toval, Ambrosio and Piattini, "A systematic review and comparison of security ontologies," in *Availability, Reliability and Security, 2008. ARES 08. Third International Conference on*, 2008, pp. 813--820.

[175] M.-A. Sicilia, E. García-Barriocanal, J. Bermejo-Higuera, and S. Sánchez-Alonso, "What are Information Security Ontologies Useful for?," 2015, pp. 51–61.

[176] S. Alqahtani, "Knowledge Modeling Survey dataset," 2018. [Online]. Available: https://github.com/isultane/KM-survey-dataset. [Accessed: 20-May-2018].

[177] S. Seedorf and F. F. I. U. Mannheim, "Applications of Ontologies in Software Engineering," in *In 2nd International Workshop on Semantic Web Enabled Software Engineering (SWESE 2006)*, 2006.

[178] D. Dermeval, J. Vilela, I. I. Bittencourt, J. Castro, S. Isotani, P. Brito, and A. Silva, "Applications of ontologies in requirements engineering: a systematic review of the literature," *Requir. Eng.*, vol. 21, no. 4, pp. 405–437, Nov. 2016.

[179] W. Kang and Y. Liang, "A Security Ontology with MDA for Software Development," in *2013 International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery*, 2013, pp. 67–74.

[180] G. Elahi, E. Yu, and N. Zannone, "A Modeling Ontology for Integrating Vulnerabilities into Security Requirements Conceptual Foundations," 2009, pp. 99–114.

[181] F. den Braber, T. Dimitrakos, B. A. Gran, M. S. Lund, K. Stolen, and J. O. Aagedal, "The CORAS Methodology," in *UML and the Unified Process*, IGI Global, 2003, pp. 332–357.

[182] R. Matulevičius, N. Mayer, H. Mouratidis, E. Dubois, P. Heymans, and N. Genon, "Adapting Secure Tropos for Security Risk Management in the Early Phases of Information Systems Development," in *Advanced Information Systems Engineering*, Berlin, Heidelberg, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 541–555.

[183] A. Souag, C. Salinesi, R. Mazo, and I. Comyn-Wattiau, "A Security Ontology for Security Requirements Elicitation," 2015, pp. 157–177.

[184] P. El Khoury, A. Mokhtari, E. Coquery, and M.-S. Hacid, "An Ontological Interface for Software Developers to Select Security Patterns," in *2008 19th International Conference on Database and Expert Systems Applications*, 2008, pp. 297–301.

[185] P. W. Singer and A. Friedman, *Cybersecurity and Cyberwar: What Everyone Needs to Know*. 2014.

[186] J. Undercoffer, A. Joshi, T. Finin, and J. Pinkston, "A Target-Centric Ontology for Intrusion Detection," in *Proceedings of the IJCAI-03 Workshop on Ontologies and Distributed Systems*, 2004, pp. 47--58.

[187] J. Undercoffer, A. Joshi, and J. Pinkston, "Modeling Computer Attacks: An Ontology for Intrusion Detection," in *Recent Advances in Intrusion Detection*, 2003, pp. 113–135.

[188] J. B. Kopena and W. C. Regli, "DAMLJessKB: A Tool for Reasoning with the Semantic Web," in *Second International Semantic Web Conference*, 2003, pp. 628–643.

[189] S. More, M. Matthews, A. Joshi, and T. Finin, "A Knowledge-Based Approach to Intrusion Detection Modeling," in *IEEE Symposium on Security and Privacy Workshops*, 2012, pp. 75–81.

[190] P. N. Mendes, M. Jakob, A. García-Silva, and C. Bizer, "DBpedia spotlight: shedding light on the web of documents," in *Proceedings of the 7th International Conference on Semantic Systems - I-Semantics '11*, 2011, pp. 1–8.

[191] Z. Syed, A. Padia, T. Finin, M. L. Mathews, and A. Joshi, "UCO: A Unified Cybersecurity Ontology," in *AAAI Workshop: Artificial Intelligence for Cyber Security*, 2016.

[192] M. Iannacone, S. Bohn, G. Nakamura, J. Gerth, K. Huffer, R. Bridges, E. Ferragut, and J. Goodall, "Developing an Ontology for Cyber Security Knowledge Graphs," in *Proceedings of the 10th Annual Cyber and Information Security Research Conference*, 2015, pp. 1–4.

[193] P. Kamongi, S. Kotikela, K. Kavi, M. Gomathisankaran, and A. Singhal, "VULCAN: Vulnerability Assessment Framework for Cloud Computing," in *2013 IEEE 7th International Conference on Software Security and Reliability*, 2013, pp. 218–226.

[194] A. Steele, "Ontological Vulnerability Assessment," in *Web Information Systems Engineering – WISE 2008 Workshops*, Berlin, Heidelberg, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 24–35.

[195] K. Srujan and K. K. G. Mahadevan, "Vulnerability Assessment In Cloud Computing," in *Proceedings of the International Conference on Security and Management (SAM)*, 2012, pp. 1–7.

[196] A. Gyrard, C. Bonnet, and K. Boudaoud, "An Ontology-Based Approach for Helping to Secure the ETSI Machine-to-Machine Architecture," in *2014 IEEE International Conference on Internet of Things(iThings), and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom)*, 2014, pp. 109–116.

[197] A. Gyrard, C. Bonnet, and K. Boudaoud, "The STAC (security toolbox: attacks &amp; countermeasures) ontology," in *Proceedings of the 22nd International Conference on World Wide Web - WWW '13 Companion*, 2013, pp. 165–166.

[198] J. A. Wang and M. Guo, "Security Data Mining in an Ontology for Vulnerability Management," in *2009 International Joint Conference on Bioinformatics, Systems Biology and Intelligent Computing*, 2009, pp. 597–603.

[199] J. A. Wang, M. Guo, H. Wang, M. Xia, and L. Zhou, "Environmental Metrics for

Software Security Based on a Vulnerability Ontology," in *2009 Third IEEE International Conference on Secure Software Integration and Reliability Improvement*, 2009, pp. 159–168.

[200] J. A. Wang, M. Guo, H. Wang, M. Xia, and L. Zhou, "Ontology-based security assessment for software products," in *Proceedings of the 5th Annual Workshop on Cyber Security and Information Intelligence Research Cyber Security and Information Intelligence Challenges and Strategies - CSIIRW '09*, 2009, p. 1.

[201] J. A. Wang, H. Wang, M. Guo, L. Zhou, and J. Camargo, "Ranking Attacks Based on Vulnerability Analysis," in *2010 43rd Hawaii International Conference on System Sciences*, 2010, pp. 1–10.

[202] J. A. Wang, L. Zhou, M. Guo, H. Wang, and J. Camargo, "Measuring Similarity for Security Vulnerabilities," in *2010 43rd Hawaii International Conference on System Sciences*, 2010, pp. 1–10.

[203] I. Kotenko, A. Chechulin, E. Doynikova, and A. Fedorchenko, "Ontological Hybrid Storage for Security Data," 2018, pp. 159–171.

[204] A. V. Fedorchenko, I. V. Kotenko, E. V. Doynikova, and A. A. Chechulin, "The ontological approach application for construction of the hybrid security repository," in *2017 XX IEEE International Conference on Soft Computing and Measurements (SCM)*, 2017, pp. 525–528.

[205] R. Montesino and S. Fenz, "Automation Possibilities in Information Security Management," in *2011 European Intelligence and Security Informatics Conference*, 2011, pp. 259–262.

[206] G. Jiang, K. Ogasawara, A. Endoh, and T. Sakurai, "Context-based ontology building support in clinical domains using formal concept analysis," *Int. J. Med. Inform.*, vol. 71, no. 1, pp. 71–81, Aug. 2003.

[207] G. Fu, "FCA based ontology development for data integration," *Inf. Process. Manag.*, vol. 52, no. 5, pp. 765–782, Sep. 2016.

[208] J. Nanda, T. W. Simpson, S. R. T. Kumara, and S. B. Shooter, "A Methodology for Product Family Ontology Development Using Formal Concept Analysis and Web Ontology Language," *J. Comput. Inf. Sci. Eng.*, vol. 6, no. 2, p. 103, 2006.

[209] L. He and Q. Wang, "Construction of Ontology Information System Based on Formal Concept Analysis," 2011, pp. 83–88.

[210] X. Bai and X. Zhou, "Development of Ontology-Based Information System Using Formal Concept Analysis and Association Rules," 2011, pp. 121–126.

[211] N. Noy and D. McGuinness, "Ontology Development 101: A Guide to Creating Your First Ontology," 2001.

[212] I. V. Krsul, "Software vulnerability analysis," Purdue University, 1998.

[213] D. Kosutic, "ISO 27001/ISO 22301 Knowledge base," *ISO 27001/ISO 22301*, 2017. [Online]. Available: https://advisera.com/27001academy/knowledgebase/. [Accessed: 14-May-2018].

[214] A. Vorobiev and Jun Han, "Security Attack Ontology for Web Services," in *Second International Conference on Semantics, Knowledge and Grid*, 2006, pp. 42–42.

[215] S. S. Alqahtani, E. E. Eghan, and J. Rilling, "SE-GPS," 2015. [Online]. Available: http://aseg.cs.concordia.ca/segps. [Accessed: 26-Sep-2017].

[216] S. O. Kuznetsov, "Stability as an Estimate of the Degree of Substantiation of Hypotheses on the Basis of Operational Similarity," *Sci. Tech. Inf. Ser. 2*, vol. 24, pp. 21–29, 1990.

[217] S. O. Kuznetsov, "On stability of a formal concept," *Ann. Math. Artif. Intell.*, vol. 49, no. 1–4, pp. 101–115, Aug. 2007.

[218] B. L. Bullough, A. K. Yanchenko, C. L. Smith, and J. R. Zipkin, "Predicting Exploitation of Disclosed Software Vulnerabilities Using Open-source Data," in *Proceedings of the 3rd ACM on International Workshop on Security And PrivacyAnalytics - IWSPA '17*, 2017, pp. 45–53.

[219] 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.

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

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

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

[223] 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].

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

[225] 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.

[226] 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.

[227] 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.

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

[229] 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.

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

[231] A. Hmood, I. Keivanloo, and J. Rilling, "SE-EQUAM - An Evolvable Quality Metamodel," in *2012 IEEE 36th Annual Computer Software and Applications Conference Workshops*, 2012, pp. 334–339.

[232] J. Z. Gao, C. Chen, Y. Toyoshima, and D. K. Leung, "Engineering on the Internet for global software production," *Computer (Long. Beach. Calif).*, vol. 32, no. 5, pp. 38–47, May 1999.

[233] F. Thung, D. Lo, and J. Lawall, "Automated library recommendation," *Proc. - Work. Conf. Reverse Eng. WCRE*, no. October, pp. 182–191, 2013.

[234] M. M. Rahman, C. K. Roy, and D. Lo, "RACK: Automatic API Recommendation Using Crowdsourced Knowledge," in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, 2016, pp. 349–359.

[235] C. Teyton, J.-R. Falleri, and X. Blanc, "Mining Library Migration Graphs," in *2012 19th*

Working Conference on Reverse Engineering, 2012, pp. 289–298.

[236] A. Hora, A. Hora, and M. T. Valente, "apiwave : Keeping Track of API Popularity and Migration," no. JANUARY, pp. 321–323, 2015.

[237] S. Raemaekers, A. van Deursen, and J. Visser, "Measuring software library stability through historical version analysis," in *2012 28th IEEE International Conference on Software Maintenance (ICSM)*, 2012, pp. 378–387.

[238] F. McCarey, M. Ó. Cinnéide, and N. Kushmerick, "Rascal: A Recommender Agent for Agile Reuse," *Artif. Intell. Rev.*, vol. 24, no. 3–4, pp. 253–276, Nov. 2005.

[239] D. L. Parnas, "Software aging," in *ICSE '94 Proceedings of the 16th international conference on Software engineering*, 1994, pp. 279–287.

[240] F. S. Foundation, "Various Licenses and Comments About Them," *GNU Project*, 2014. .

[241] L. Xavier, A. Brito, A. Hora, and M. T. Valente, "Historical and impact analysis of API breaking changes: A large-scale study," in *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2017, pp. 138–147.

[242] S. Raemaekers, A. Van Deursen, and J. Visser, "Semantic versioning versus breaking changes: A study of the maven repository," *Proc. - 2014 14th IEEE Int. Work. Conf. Source Code Anal. Manip. SCAM 2014*, pp. 215–224, 2014.

[243] O. Seneviratne, L. Kagal, D. Weitzner, H. Abelson, T. Berners-Lee, and N. Shadbolt, "Detecting creative commons license violations on images on the world wide web," *WWW2009, April*, 2009.

[244] A. Hmood, Philipp Schugerl1, J. Rilling, and Philippe Charland, "OntEQAM – A Methodology for Assessing Evolvability as a Quality Factor in Software Ecosystems," in *Defence R&D Canada - Valcartier, Valcartier QUE (CAN)*, 2010, p. 8.

[245] J. A. McCall, P. K. Richards, and G. F. Walters, "Factors in Software Quality. Volume I. Concepts and Definitions of Software Quality," 1977.

[246] A. Bergel, S. Denier, S. Ducasse, J. Laval, F. Bellingard, P. Vaillergues, F. Balmas, and K. Mordal-Manet, "SQUALE - Software QUALity Enhancement," in *2009 13th European Conference on Software Maintenance and Reengineering*, 2009, pp. 285–288.

[247] H. Kagdi, S. Yusuf, and J. I. Maletic, "Mining sequences of changed-files from version histories," in *Proceedings of the 2006 international workshop on Mining software repositories - MSR '06*, 2006, p. 47.

[248] H. Kagdi, M. L. Collard, and J. I. Maletic, "Comparing Approaches to Mining Source Code for Call-Usage Patterns," in *Fourth International Workshop on Mining Software Repositories (MSR'07:ICSE Workshops 2007)*, 2007, pp. 20–26.

[249] T. Kamiya, S. Kusumoto, and K. Inoue, "CCFinder: a multilinguistic token-based code clone detection system for large scale source code," *IEEE Trans. Softw. Eng.*, vol. 28, no. 7, pp. 654–670, Jul. 2002.

[250] Y. Zhang, R. Witte, J. Rilling, and V. Haarslev, "Ontological approach for the semantic recovery of traceability links between software artefacts," *IET Softw.*, vol. 2, no. 3, p. 185, 2008.

[251] I. Keivanloo, C. Forbes, J. Rilling, and P. Charland, "Towards sharing source code facts using linked data," *Proceeding 3rd Int. Work. Search-driven Dev. users, infrastructure, tools, Eval. - SUITE '11*, pp. 25–28, 2011.

[252] M. F. Bertoa, A. Vallecillo, and F. García, "An Ontology for Software Measurement," in *Ontologies for Software Engineering and Software Technology*, Springer Berlin Heidelberg, 2006, pp. 175–196.

[253] R. Robbes, M. Lungu, and D. Röthlisberger, "How do developers react to API deprecation?," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering - FSE '12*, 2012, p. 1.

[254] B. E. Cossette and R. J. Walker, "Seeking the Ground Truth: A Retroactive Study on the Evolution and Migration of Software Libraries," *Proc. ACM SIGSOFT 20th Int. Symp. Found. Softw. Eng.*, p. 55:1--55:11, 2012.

[255] P. Kapur, B. Cossette, and R. J. Walker, "Refactoring references for library migration," *ACM SIGPLAN Not.*, vol. 45, no. 10, p. 726, 2010.

[256] L. A. Zadeh, "The concept of a linguistic variable and its application to approximate reasoning-III," *Inf. Sci. (Ny).*, vol. 9, no. 1, pp. 43–80, Jan. 1975.

[257] I. E. Commission, "Programmable Controllers - Part 7: Fuzzy Control Programming," 2000.

[258] P. Cingolani and J. Alcala-Fdez, "jFuzzyLogic: a robust and flexible Fuzzy-Logic inference system language implementation," in *2012 IEEE International Conference on Fuzzy Systems*, 2012, pp. 1–8.

[259] I. Samoladas, G. Gousios, D. Spinellis, and I. Stamelos, "The SQO-OSS Quality Model: Measurement Based Open Source Software Evaluation," in *Open Source Development, Communities and Quality*, Boston, MA: Springer US, 2008, pp. 237–248.

[260] B. M. Kuhn, A. K. Sebro, and D. Gingerich, "Chapter 10 The Lesser GPL," *Free Software Foundation & Software Freedom Law Center*, 2016. .

[261] V. del Bianco, L. Lavazza, S. Morasca, and D. Taibi, "Quality of Open Source Software: The QualiPSo Trustworthiness Model," 2009, pp. 199–212.

[262] T. Boland, C. Cleraux, and E. Fong, "Toward a Preliminary Framework for Assessing the Trustworthiness of Software," 2010.

[263] R. Jagarlamudi, Jagadeesh and Daum III, Hal and Udupa, "Incorporating lexical priors into topic models," in *Proceedings of the 13th Conference of the European Chapter of the Association for Computational Linguistics*, 2012, pp. 204--213.

[264] M. I. Blei, David M and Ng, Andrew Y and Jordan, "Latent dirichlet allocation," *J. Mach. Learn. Res.*, vol. 3, pp. 993--1022, 2003.

[265] S. S. Alqahtani and J. Rilling, "An Ontology-Based Approach to Automate Tagging of Software Artifacts," in *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, 2017, pp. 169–174.