THESIS

A HEURISTIC-BASED APPROACH TO AUTOMATICALLY EXTRACT PERSONALIZED ATTACK GRAPH RELATED CONCEPTS FROM VULNERABILITY DESCRIPTIONS

Submitted by

Subhojeet Mukherjee

Department of Computer Science

In partial fulfillment of the requirements

For the Degree of Master of Science

Colorado State University

Fort Collins, Colorado

Fall 2017

Master's Committee:

Advisor: Indrajit Ray

Indrakshi Ray
Zinta Byrne

ABSTRACT

A HEURISTIC-BASED APPROACH TO AUTOMATICALLY EXTRACT PERSONALIZED ATTACK GRAPH RELATED CONCEPTS FROM VULNERABILITY DESCRIPTIONS

Computer users are not safe, be it at home or in public places. Public networks are more often administered by trained individuals who attempt to fortify those networks using strong administrative skills, state-of-the-art security tools and meticulous vigilance. This is, however, not true for home computer users. Being largely untrained they are often the most likely targets of cyber attacks. These attacks are often executed in cleverly interleaved sequences leading to the eventual goal of the attacker. The Personalized Attack Graphs (PAG) introduced by Ubranska et al. [24, 25, 32] can leverage the interplay of system configurations, attacker and user actions to represent a cleverly interleaved sequence of attacks on a single system. An instance of the PAG can be generated manually by observing system configurations of a computer and collating them with possible security threats which can exploit existing system vulnerabilities and/or misconfigurations. However, the amount of manual labor involved in creating and periodically updating the PAG can be very high. As a result, attempt should be made to automate the process of generating the PAG. Information required to generate these graphs are available on the Internet in the form of vulnerability descriptions. This information is, however, almost always written in natural language and lacks any form of structure. In this thesis, we propose an unsupervised heuristic-based approach which parses vulnerability descriptions and extracts instances of PAG related concepts like system configurations, attacker and user actions. Extracted concepts can then be interleaved to generate the Personalized Attack Graph.

ACKNOWLEDGEMENTS

I would like to thank my advisor Dr.Indrajit Ray for giving me every opportunity to showcase my abilities. Dr. Ray's contributions to my life, education and this thesis is more than words can describe.

I would like to thank my grandmother Smt. Sipra Ghatak, mother Mrs. Soma Mukherjee, father Mr. Sanjoy Mukherjee and wife Antara Chakraborty for the consistent support and encouragement. For 'MA' and 'DIDA', this is one step of reward for all that you have done or are still doing for me. For 'BABA', you taught me the maths and physics which forms the base of all my thoughts. For Antara, only a married graduate student knows how much sacrifice their spouse makes just to see this 100 page document. I could not thank you more, you are the best.

For Kush and Ibrahim, thanks a lot for spending a large chunk of your valuable time for tagging my datasets. For Sachini, thanks a lot for doing so much in that first paper.

This material is based upon work supported by the National Science Foundation under Grant No. 0905232.

DEDICATION

I would like to dedicate this thesis, to my co-advisor Dr.Adele Howe; "Dr.Howe I finally defended". Thanks for believing in a student who hardly knew C++ back then. Thank you for making me a better writer, thinker and researcher. Also, to 'DIDA' and 'MA', for being the pillars that I stand on.

TABLE OF CONTENTS

ABSTRACT ii
ACKNOWLEDGEMENTS iii
DEDICATION iv
LIST OF TABLES vii
LIST OF FIGURES viii

Chapter 1 Introduction and Problem Description 1
1.1 The Big Picture 1
1.2 Personalized Attack Graph (PAG) 3
1.2.1 PAG Overview 3
1.2.2 Constructing the PAG 5
1.3 Challenges in Extracting the PAG Concepts Automatically 6
1.4 Thesis Contribution and Organization 8

Chapter 2 Related Work 10

Chapter 3 Background Knowledge 13
3.1 Stanford Parser 14
3.2 Stanford Typed Dependency Representation 16
3.3 Wordnet 18
3.4 SentiWordnet 19
3.5 CVE-Details 20

Chapter 4 Formulating the PAG concepts 22
4.1 Pre-processing the Input Text 22
4.2 Extracting Software Names, Versions and Modifiers (Component 1) 24
4.2.1 Creating a Sentence-Bundle 26
4.2.2 Getting Software Names from CVE-DETAILS 28
4.2.3 Identifying Software Names in Text 30
4.2.4 Cleaning up 35
4.2.5 Identifying Versions 35
4.2.6 Version Decisioning 42
4.2.7 Identifying Modifiers 46
4.2.8 Post-Processing 48
4.3 Extracting Attacker Actions, User Actions and Post-Condi tions (Component 2) 48
4.3.1 Creating a Sentence-Bundle 50
4.3.2 Identifying Human Actors 54
4.3.3 Classifying Indirect Actors as Humans 57
4.3.4 Assigning Attributes to Actors 59
4.3.5 Determining the Polarity of an Actor 59

4.3.6 Sectioning Into Attacker Actions, User Actions and Post-Conditions . . 61
4.3.7 Cleaning up 66

Chapter 5 Evaluation and Discussion 69
5.1 Evaluation Preliminaries 69
5.1.1 DataSets 69
Joshi Corpus 70
Our Corpus 72
5.1.2 Evaluation Metrics 72
5.2 Evaluating on the Joshi Corpus 74
5.3 Evaluating on Our Corpus 76
5.4 Effectiveness of the Heuristics 77
5.4.1 Step 1 78
5.4.2 Step 2 79
5.4.3 Step 3 80

Chapter 6 Conclusion and Future work 82

Bibliography 84

LIST OF TABLES

1.1 PAG concepts 6

3.1 Complete Set of Syntactic tags supported by the Penn Treebank Project 16
3.2 Complete Set of Relations provided by the Stanford Typed Dependency Representation 17
3.3 Sentiment Score Calculation for “Impulsive" 20

4.1 Grammatical Dependency List for Vulnerability Description [CVE-2010-0483] (dList) 53
4.2 Types of Attributes Assigned to Each Actor 58

5.1 Joshi Corpus Classes vs PAG Concepts 70
5.2 Joshi Corpus Concept Instance Distribution 71
5.3 Our Corpus Concept Instance Distribution 71
5.4 Results for Experiments Run on the Joshi Corpus 74
5.5 Results for Experiments Run on Our Corpus 76
5.6 NVD Precision and Recall P-Values [Joshi and Our Corpus] 78
5.7 NVD vs Bulletins and Blogs Precision and Recall Samples for Wilcoxon Test [Joshi Corpus] 79
5.8 NVD vs Bulletins and Blogs Precision and Recall P-Values [Joshi Corpus] 80
5.9 NVD vs Other Semi-Structured Precision and Recall Samples for Wilcoxon Test [Our Corpus] 80
5.10 NVD vs Bulletins and Blogs Precision and Recall P-Values [Joshi Corpus] 80

LIST OF FIGURES

1.1 NVD Vulnerability Growth Statistics . . . 1
1.2 Sample Personalized Attack Graph . . . 5
1.3 National Vulnerability Database [CVE-2010-0483] . . . 7

2.1 PACE Bootstrapping cycle [17] . . . 11

3.1 An Example Parse Tree generated by the Stanford Parser . . . 15
3.2 An Example Typed Dependency List . . . 18
3.3 Software Name Repository at CVE-DETAILS . . . 21

4.1 Architecture of the PAG Concept Extractor . . . 23
4.2 An example of vulnerability description pre-processing . . . 23
4.3 Component 1 Workflow Diagram . . . 25
4.4 Forming a Sentence-Bundle Tuple . . . 29
4.5 Results from CVE-DETAILS . . . 30
4.6 Selection of Software . . . 34
4.7 Identifying Versions . . . 39
4.8 Decisioning in Assigning Versions . . . 43
4.9 Final Output . . . 47
4.10 Component 2 Workflow Diagram . . . 49
4.11 Syntax Tree for Vulnerability Description [CVE-2010-0483] (*synTree*) . . . 52
4.12 Identifying Human Actors . . . 56
4.13 Attributes Assigned to each Actor . . . 60
4.14 Determining the Type of Human Actor . . . 63
4.15 Segregating Actions and Post-Conditions . . . 67
4.16 Clean up . . . 68

5.1 Comparative Evaluation of Precision and Recall Scores [Joshi Corpus] . . . 75
5.2 Comparative Evaluation of Precision and Recall Scores [Our Corpus] . . . 77

Chapter 1

Introduction and Problem Description

1.1 The Big Picture

As technology is progressing, computer systems are becoming more and more susceptible to attacks from malicious adversaries. National Vulnerability Database (NVD) provides a bar-chart representation (Fig. 1.1) [1] of the number of computer security vulnerabilities each year, from 1988 to 2014.

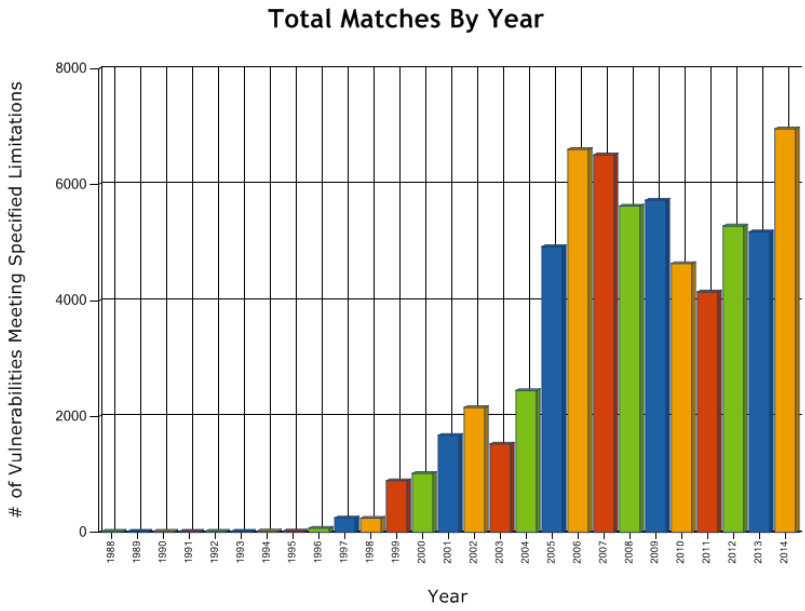


Figure 1.1: NVD Vulnerability Growth Statistics

The growing security needs demand enhanced vigilance from the system administrators and personal computer users [29] to, among other activities, identify vulnerabilities and patch vulnerable software. Vulnerability scanners like OpenVAS (`http://www.openvas.org`) and Nessus

[1] Generated by querying NVD statistics generator [`http://web.nvd.nist.gov/view/vuln/statistics`] with empty string parameters

(http://www.tenable.com/products/nessus) help in identifying isolated vulnerabilities on both networked and standalone systems. However, exploiting a single vulnerability might not always lead to the goal of the attacker. Often a group of vulnerabilities can be exploited in a cleverly interconnected sequence to meet the malevolent intents of the attacker. Vulnerability scanners like OpenVAS and Nessus do not exhibit features that formally represent the interactions between vulnerabilities and threats. This problem is solved to a great extent by making use of Attack Trees (AT) and Attack Graphs (AG) [12, 30, 34]. ATs and AGs are data structures which are used to relate different security vulnerabilities and capture how they could be exploited in a systematic manner to cause security compromises. Traditionally attack graphs and attack trees are used to analyze attack scenarios for networked systems [8, 21–23] and do not focus on interactions of vulnerabilities on a single computer. The Personalized Attack Graph (PAG) [24, 25, 32] is a morphed representation of the traditional attack graph that is suited to analyze threats on a single system. Apart from formally introducing the PAG in [24, 25, 32], Urbanska et al. [31] attempt to extract information from the web (National Vulnerability Database (NVD) [2]) that can be used to build the PAG. However, their approach makes use of syntactic patterns observed in information published by NVD and is thus suited to extracting PAG concepts from NVD only. In order to extract concepts from more than one vulnerability database (including NVD) and build the PAG as defined in [24, 25, 32], we need to construct a system which is independent of the syntactic patterns observed in information published by any particular vulnerability database. In this thesis, we leverage this idea by presenting an approach that makes use of grammatical connotations of words and phrases in the English language to parse information obtained from the web and extract concepts which can then be used to build the PAG.

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

1.2 Personalized Attack Graph (PAG)

1.2.1 PAG Overview

A Personalized Attack Graph (PAG) [24,25,32] is a graphical representation of the interactions between vulnerabilities existing on a system and actions performed by users and attacker which lead to a successful compromise of the system. In general, the PAG can be associated with four different concepts. Discussed below are the four concepts with examples.

Concept 1 (Attacker Actions)**.** *Attacker Actions are propositions that represent the operations performed by an attacker to trigger a security compromise. Attacker actions are often, if not always, aided by actions performed by users and existing vulnerabilities on a system. Attacker actions are formally represented as nodes in the PAG and can be either true or false with some probability of success. More like these are examples:*

- *Sending crafted documents to leverage faults in the system*
- *Enticing users to visit malicious website*

Concept 2 (User Actions)**.** *User Actions are propositions that represent the operations performed by a user resulting in a security compromise. User actions are often, if not always, influenced by attacker actions. User actions are formally represented as nodes in the PAG and can be either true or false with some probability of success. The probability of a User Action is calculated using user specific features like level-of-confidence in performing security related tasks and perceived benefits of risky actions. More like these are examples:*

- *Pressing a key which might trigger the installation of a malicious executable file*
- *Clicking on a link, which leads to a phishing website.*

Concept 3 (Software and Versions)**.** *Software and Versions are propositions that represent vulnerable versions of softwares which can be attacked to cause a system compromise. The presence of a vulnerable software is not enough to guarantee the successful execution of an attack. The vulnerability in the software needs to be exploited by actions from either users or attackers or both.*

Software and Versions are formally represented as nodes in the PAG and can be either true or false with some probability of success. More like these are examples:

- *vBulletin 4.4.2*
- *TableField module 7.x-2.x*

In order to fully express the concept of *Software and Versions* we break it down into 3 separate concepts namely, *Software Names*, *Versions* and *Modifiers*. While *Software Names* and *Versions* are self-explanatory, *Modifiers* are quantifiers which express the range of *Versions* susceptible to an attack. Security descriptions often use phrases like “2x before 2.3" to denote vulnerable versions. A *Modifier* [13] in this example denotes all version of a software with the prefix “2" and below 2.3 (2.0, 2.1 etc.). Although in section 4, we extract *Software Names*, *Versions* and *Modifiers* separately, throughout this thesis we refer to them using a single concept *Software and Versions*.

Concept 4 (Post-Conditions)**.** *Post-Conditions are propositions that represent the impact of a successful attack. Post-Conditions are generally intermediate nodes in a PAG that can be either true or false with some probability of success. More like these are examples:*

- *Causing arbitrary code to run on a user's machine*
- *Enticing users to visit malicious website*

An example of a Personalized Attack Graph is shown in Fig. 1.2. The four PAG concepts constituting the instance of the PAG shown in Fig. 1.2 are tabulated in Table 1.1. The PAG essentially represents the interplay of the four concepts described above and is specific to a given system. The goal node represents the eventual goal of the attacker (e.g. “Denial of service"). The sub-goals represent the effects individual attacks launched on the system (e.g. “Executing arbitrary code"). Predecessors of the goal node signify various activities that must occur in the home computer systems to trigger the goal or the final exploit. A set of such activities in the same level of the tree might happen in conjunction (“AND") or disjunction (“OR") with one another and thereby pave the path towards the execution of a different node at a higher level (lower level

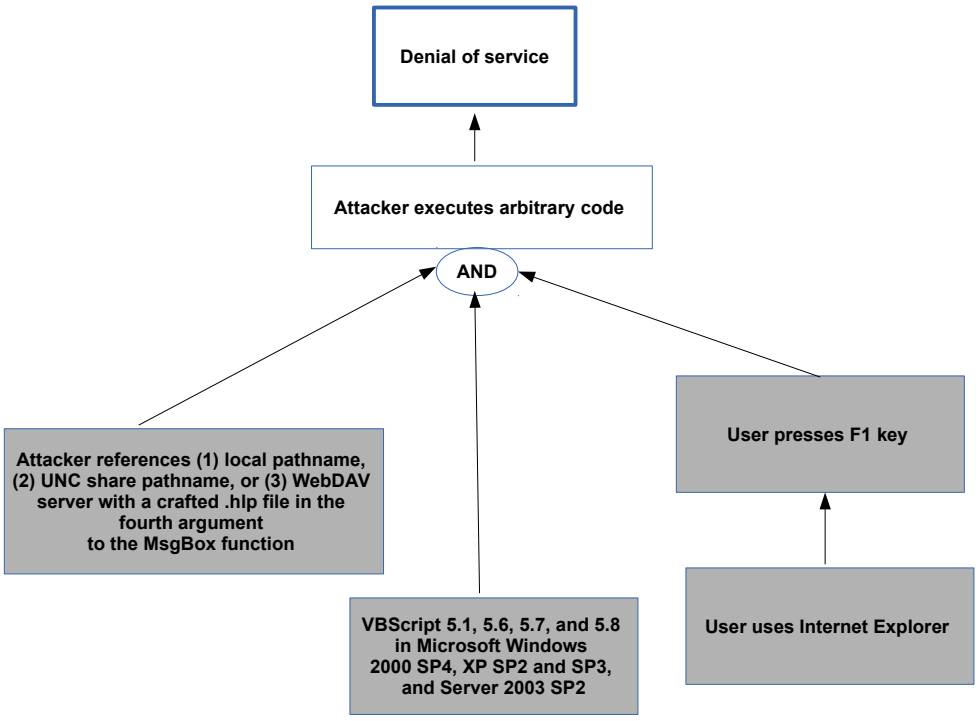


Figure 1.2: Sample Personalized Attack Graph

in terms of depth from the root). A node of the graph in Fig.1.2 maps to exactly one of the four concepts in a many-to-one manner (see Table 1.1). This implies that multiple PAG nodes can map to a single concept, but not vice-versa. Because a PAG is composed of multiple instances of these concepts, we cumulatively refer to them as the *PAG concepts*. An arc in the PAG represents the sequence of propositions, all of which need to be true for a successful system compromise.

1.2.2 Constructing the PAG

Constructing the PAG for a specific computer requires identifying all instances of the PAG concepts valid for that computer. One approach to that can be to scan the computer using vulnerability scanners and manually identify combinations of *Sofware and Versions*, *Attacker actions*, *User Actions* and *Post-Conditions* which can lead to a system compromise. This approach demands intensive manual labor and time. Moreover, because home computer users are largely untrained [1], this task becomes even more difficult for them. A possible solution can be to generate the PAG

Table 1.1: PAG concepts

Concept	Content	Shown In
Software and Versions	"VBScript 5.1, 5.6, 5.7, and 5.8 in Microsoft Windows 2000 SP4, XP SP2 and SP3, and Server 2003 SP2"	shaded-box
Attacker Action	"references a (1) local pathname, (2) UNC share pathname, or (3) WebDAV server with a crafted .hlp file in the fourth argument to the MsgBox function"	shaded box
User Action	"uses Internet Explorer", "presses F1 key"	shaded box
Post-Condition (including eventual goal)	"executes arbitrary code", "Denial of service"	plain box (goal shown in thick bordered box)

in an automated fashion. Roughly, the automated construction of the PAG involves the following steps:

1. Scan the system for existing vulnerabilities
2. Obtain textual descriptions of the PAG concepts related to each observed vulnerability. These textual descriptions can then be used to represent the propositional values for the nodes of the PAG. For instance, the PAG in Fig. 1.2 can be constructed by obtaining textual representations of the PAG concepts shown in the second column of Table 1.1.
3. Establish hierarchical relationships among the nodes.

Step 1 can be achieved easily using modern vulnerability scanners. Step 2 requires obtaining information pertaining to the PAG concepts and is the goal of this thesis. Step 3 requires further analysis of the extracted descriptions and is out of the scope of this thesis.

1.3 Challenges in Extracting the PAG Concepts Automatically

As discussed previously, the goal of this thesis is to obtain instances of the PAG concepts in an automated fashion. This task is, however, accompanied by its own challenges.

Firstly, textual descriptions of the PAG concepts need to obtained from somewhere. Fortunately, this information is provided in plenty on the web. Fig. 1.3 shows one such web-page

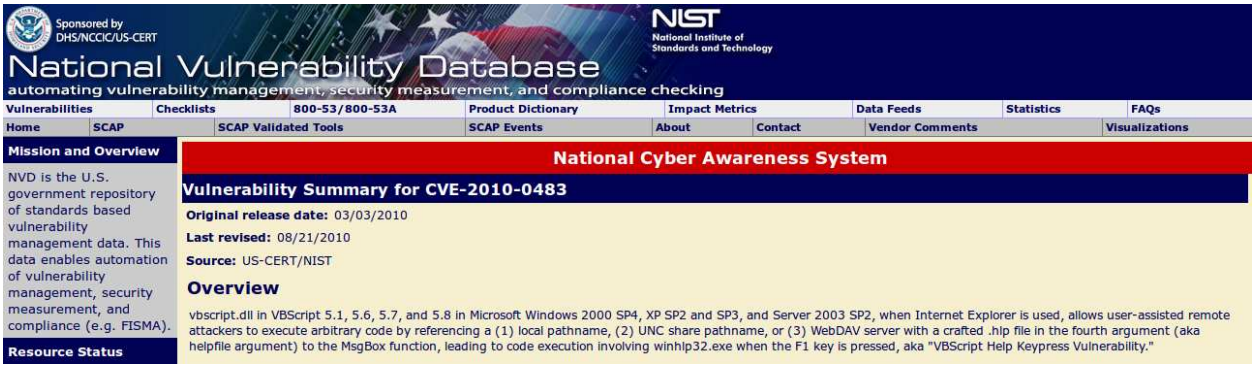


Figure 1.3: National Vulnerability Database [CVE-2010-0483]

hosted by the National Vulnerability Database (NVD). This page presents detailed information on the vulnerability referred to by the identifier CVE-2014-2489[3]. The block of text provided under the “Overview" section in Fig. 1.3 is often referred to as a *vulnerability description*. The PAG concepts are embedded in these vulnerability descriptions and in order to construct the PAG we need to extract these concepts from the description. Table 1.1 shows the PAG concepts that can be extracted from the vulnerability description provided in Fig. 1.3 and used to build the PAG from Fig. 1.2.

Secondly, once the vulnerability descriptions are obtained they need to be parsed in order to extract the PAG concepts. However, vulnerability descriptions are published and updated by human beings. Subsequently, they are written in natural language and do not follow a pre-defined structure or schema. The unstructured nature of the descriptions poses two serious issues for the automated concept extraction process. Firstly, it needs to be verified whether a particular concept is present in vulnerability description. And secondly, if it is present, which portion(s) of the descriptions correspond to instances of that concept.

[3]The CVE identifier is a unique identifier assigned to newly discovered vulnerabilities. The MITRE Corporation (`http://www.cve.mitre.org/cve/identifiers/index.html`) is primarily responsible for providing these identifiers

1.4 Thesis Contribution and Organization

In this thesis, we aim to extract the four PAG concepts (*Software and Versions*, *Attacker actions*, *User Actions* and *Post-Conditions*) from vulnerability descriptions. Although the problem is similar to the named entity recognition (NER) paradigm, the entities extracted in a typical NER application (names, people, organizations etc.) do not coincide with the PAG concepts [19]. Previous works [13, 17, 19] make use of supervised/semi-supervised machine learning techniques to extract cyber-security related concepts (the concepts they extract are not necessarily the same as ours). However, we believe supervised learning techniques are not perfectly suited for PAG concept extraction. This is because supervised learning techniques require a substantial amount of training data and there is no publicly available annotated dataset for PAG concepts. Even if such a dataset is generated, it requires further effort to maintain and update it if the source information changes. In view of these issues posed by supervised learning techniques, we propose an alternative unsupervised heuristic-based approach to parsing security related descriptions written in natural language. In particular, we try to make inferences based on grammatical patterns and parts-of-speech that are commonly used in English. For this purpose, we make use of publicly available tools [2, 7, 14, 18], which support automated annotation and reasoning of different parts-of-speech occurring in natural language. However, in this process we face two primary issues. Firstly, not all judgments can be made using natural language tools. For example, to the best of our knowledge, there does not exist an NLP tool or a publicly accepted heuristic which can establish a concrete relation between a subject of a sentence and a set of verbs from the same sentence. Secondly, most of these NLP tools are trained on manually annotated corpora that are not specifically tailored for computer security. Thus the accuracy of these tools, in annotating security related concepts, might impact the final results we obtain. To counter these two issues, we base our heuristics on semantic patterns commonly observed in vulnerability descriptions published by the National Vulnerability Database (NVD). For example, thoroughly observing vulnerability descriptions from NVD reveal that *Post-Conditions* are generally found as substrings in the object part of a sentence.

The rest of the thesis is organized as follows. Chapter 2 delves into some of the previous work done in the field of cyber-security related concept extraction. Although these works do not necessarily focus on extracting the PAG concepts, they help in familiarizing the reader with previous research done in the same line as ours. In Chapter 3, we provide a detailed description of the data repository and NLP tools used for the information extraction techniques employed in this thesis. In Chapter 4, we introduce a set of algorithms that extract PAG concepts from vulnerability descriptions in an unsupervised heuristic-based manner. Chapter 5 provides an evaluation of the efficiency of our approach in terms of standard metrics like precision and recall [15] followed by a comparison of the scores to that obtained by Joshi et al. [13]. Using our evaluation we also attempt to answer the cardinal question raised above. Finally in Chapter 6, we summarize the contributions made in this thesis followed by the prospective future work.

Chapter 2

Related Work

Till date, research in the field of cyber-security related concept extraction has been limited. Amongst the few works done in this field of research, only one focuses on extracting concepts similar to ours. Then again concepts like *User Action* have been completely overlooked. This is because previous works were not motivated by the idea of extracting concepts related to the Personalized Attack Graphs. Nevertheless, the literature review done in this chapter helps the reader to get acquainted with the field of cyber-security related concept extraction.

In their work, Roschke et al. [26, 27] attempt to solve a problem similar to ours. They parse security descriptions and extract entities out of it to build attack graphs. Although they present a comparison of 10 vulnerability databases based on the entities which can be extracted from them, they eventually choose NVD as their main source of information. The entity extraction process is facilitated by creating “reader plugins". Reader plugins make use of common syntactic patterns and phrases used in vulnerability descriptions. As an example, NVD often uses the phrase “execute arbitrary code" to describe a post-condition of an attack. The authors propose a reader plugin for each vulnerability information source. They also propose the concept of “writer plugins" which transform the extracted entities to forms which can be read by various applications like attack graph generators or vulnerability analysis tools. The main problem with this approach is the use of a distinct reader plugin for each vulnerability database. Because reader plugins make use of common patterns and phrases, users need to identify patterns in unstructured texts across all vulnerability databases that are candidates for the extraction process. This not only becomes tedious but also for many vulnerability databases it is almost impossible to find syntactic patterns in descriptions. A separate problem can occur with the maintenance of the reader plugins. If the patterns change over the course of time, the reader plugins need to be adjusted to match the new patterns. Urbanska et al. [31] use a similar approach, but theirs is scoped to extracting PAG concepts from NVD only. This work suffers from the same drawbacks as observed by Roschke et al. [26, 27].

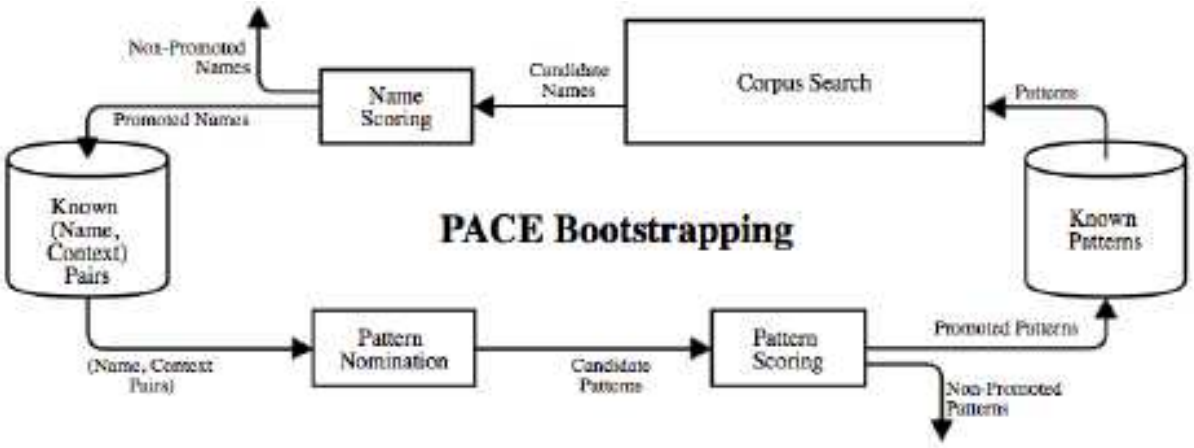


Figure 2.1: PACE Bootstrapping cycle [17]

PACE [17] is a bootstrapping (semi-supervised) algorithm which tries to extract 4 security related concepts (*Exploit Effect*, *Software Name*, *Vulnerability Potential Effects*, *Vulnerability Category*) from multiple vulnerability databases. The algorithm requires a small set of known entity-context pairs, known patterns and an input corpus as input. The algorithm then proceeds using a cyclic approach similar to traditional bootstrapping [20] (Fig. 2.1). Known entity-context pairs are expressed using entity names and 5 word prefix/suffix. For example,

$$\underbrace{\text{“}exploits\ ...\ flaws\ in\text{”}}_{\text{prefix}} + \underbrace{\text{“}Android\text{”}}_{\text{entity name}} \underbrace{\text{“}and\ borrows...Windows\text{”}}_{\text{suffix}}$$

Once the known entity-context pairs are provided, the algorithm learns new known patterns by comparing them to known entity-context pairs and choosing the best-ranked patterns from ones produced by the comparisons (Fig. 2.1). Patterns are expressed using the triple [*prefix*, *name*, *suffix*]. The known patterns are then used to search the input corpus and generate candidate entity names. Candidates are then scored and the best are added to the set of known entity-names. This process continues in a cycle, as shown in Fig. 2.1. The number of such cycles is determined by another input to the algorithm. Because patterns are created from a small set of known entity-context pairs, the patterns are generally accurate and take much less time to be generated than traditional bootstrapping processes which scan the whole corpus to generate patterns. However, PACE suffers from a few drawbacks. Firstly, the algorithm requires a very large corpus, across multiple vulnerability databases as input. And secondly, although syntactic patterns are hard to

find in vulnerability descriptions, even if patterns exist, they vary largely from one database to another. As a result, syntactic patterns extracted from one database is hardly useful to extract patterns from another.

Joshi et al. [13] demonstrate a machine learning based algorithm to extract 7 security related concepts: software (e.g., Adobe Reader 10.2), network terms (e.g., HTTP, SSL), attack cause (e.g., buffer overflow), attack consequence (e.g., denial of service), file name (e.g., index.php), hardware (e.g., Cisco Router), and modifiers (e.g.version before 10.2). In particular, they make use of a Condition Random Field Classifier [33] with 7 feature sets to identify the concepts. The training dataset used by them consisted of vulnerability descriptions from 30 security blogs, 240 CVE descriptions (same as extracted from NVD) and 80 Security Bulletin entries from Abode and Microsoft. Once the concepts are extracted, they are mapped to DBPedia [3] resources and expressed as RDF triples using classes from an existing IDS ontology[4]. The extracted concept triples are accompanied with RDF triples formed by absorbing NVD XML feeds. Although some of the concepts extracted in this work are quite similar to the PAG concepts, the authors do not spot the *User Action* entity from the vulnerability texts. Moreover, since this work uses supervised machine learning techniques, it requires sufficient domain-specific training data. To the best of our knowledge, no such dataset exists for PAG related concepts.

Mulwad et al. [19] make of use Wikitology [11] which is an “off-the-shelf" knowledge base of information from Wikipedia, DBpedia [3] etc to extract information related to vulnerabilities, threats and attacks. Computer security related concepts are extracted by querying the Wikitology knowledge base against the text descriptions and the returned concepts are pruned to a set whose elements belong to the taxonomies under the Wikipedia category [5] “Computer_security_exploits". The extracted concepts are then mapped to the IDS OWL ontology and DBPedia resources, similar to the previous work [13] described in this section.

[4] https://github.com/ArnavJoshi/IDSOntology/blob/master/IDSv2.0.1.owl

[5] https://en.wikipedia.org/wiki/Special:CategoryTree

Chapter 3

Background Knowledge

The PAG concepts (*Software and Versions*, *Attacker Actions*, *User Actions*, *Post Conditions*) are embedded in vulnerability descriptions available on the web. In order to formulate the PAG, we need to extract these concepts from vulnerability descriptions written in natural language. In Chapter 4 we propose an unsupervised heuristic-based approach to extract the PAG concepts automatically. Since vulnerability descriptions are free texts expressed in natural language, in order to devise heuristics which exploit grammatical features of these free texts we need to perform natural language processing on them. This can be done using cleverly crafted custom-created algorithms. Fortunately, prior research and development in the field of natural language processing and computer security have led to the fabrication of tools and data repositories which can be used for this purpose. Listed below are some of the basic requirements of our work along with the tools and data repositories which satisfy them.

- A large part of our solution makes use of rules created by thoroughly observing the general grammatical structures in English sentences. In particular we observe two fundamental aspects of English grammar: *parts-of-speech* (like noun, verb, adjective etc.) of a word in the context of the sentence it belongs to and *grammatical dependencies* between words (eg. “attacker" is the subject of the verb “compromise" in the sentence “attacker compromised the machine"). The *Stanford Parser* [14] and the *Stanford Typed Dependency Representation* [5,7] allow us to ascertain the parts-of-speech and inter-word dependencies in a sentence respectively.

- Although Stanford Typed Dependency Representation assists in finding the subject of a given verb, it does not give enough information about the subject. In particular, we seek to find whether the subject of a verb in a sentence is in the form of a *human*. For example, in the sentence “attacker compromised the machine", the subject “attacker" is a human. We make

use of the popular lexicon *Wordnet* [10,18] to deduce more information like this about actors in a sentence.

- A computer, in general, is not intelligent enough to semantically distinguish between words like “attack" and “accept" as actions with malicious and benign intents respectively. *SentiWordNet* [2,9] is a publicly available tool which can deduce the polarity of a word as negative (malicious) or positive/neutral (benign). Being able to distinguish between the polarity of words allows us to categorize actors in a sentence as malicious or benign.

- Finally, software names are mostly represented as proper nouns in a vulnerability description. However, the opposite might not be true, i.e. all proper nouns are not software names. Thus after identifying proper nouns, using the *Stanford Parser*, we need to validate whether they refer to popular software names. For this purpose, we match the identified proper nouns to popular software names obtained from the software repository of the vulnerability database *CVE-DETAILS*.

The rest of this chapter is subdivided into sections, each of which provides detailed information about the Stanford Parser, the Stanford Typed Dependency Representation, WordNet, SentiWordNet and the CVE-DETAILS vulnerability database respectively. It is to be noted that because these tools are designed to work on English sentences written in natural language, the examples used in this chapter are based on routinely used English sentences and not vulnerability descriptions.

3.1 Stanford Parser

The Stanford Parser [14], uses probabilistic context-free grammars(PCFG). PCFG bases on the use of the terminal and non-terminal symbols in POS tagging. Non-terminal symbols are those which can have a branching factor greater than 0, i.e. they have a child node. For example, the POS tag `S`(sentence/starting symbol) can have a set of children denoted by NP, VP. On the other hand, terminal symbols do not have any children. For example, in the English vocabulary, a word like “man" would appear as the leaf node of a parse tree and hence could be considered as a

```
Sentence : The user
pressed the f1 key.

Parse Tree:

(ROOT
  (S
    (NP (DT The) (NN
user))
    (VP (VBD pressed)
      (NP (DT the) (JJ
f1) (NN key)))
    (. .)
   )
)
```

Figure 3.1: An Example Parse Tree generated by the Stanford Parser

terminal symbol. A derivation is a rule using which each non-terminal symbol in a sentence can be broken down into a combination non-terminal or terminal symbols. For example, $< VP >\rightarrow< VB >< NP >$, where the VP on the left-hand side is the parent node and the nodes on the right are children. This scenario is also called context-free grammar, where the single symbol on the right can be rewritten using the rule for which it is the left-hand argument, without any contextual information related to it. For example, $< VP >\rightarrow< VB >< NP >$ can be rewritten as $< VP >\rightarrow< worked >< NP >$, where VB is replaced using the rule $< VB >\rightarrow< worked >$. Typically the rule *Left Most Derivation* suggests rewriting the leftmost non-terminal symbol on the right-hand side of a derivation with the rule that symbol is applicable to. However, many times a single non-terminal symbol can be segregated into multiple distinct combinations of terminal and non-terminal symbols. Under such scenarios, the sentence can produce multiple parse trees. For example, the non-terminal symbol $< VP >\rightarrow< VB >< NP >$ can also be written as $< VP >\rightarrow< VP >< PP >$. This introduces a chance factor and hence leads to multiple left-most derivation trees for a single sentence. PCFG aims to return the parse tree which has the highest chance to represent a particular sentence.

Table 3.1: Complete Set of Syntactic tags supported by the Penn Treebank Project

Tag	Parts-of-Speech
ADJP	Adjective phrase
ADVP	Adverb phrase
NP	Noun phrase
PP	Prepositional phrase
S	Simple declarative clause
SBAR	Clause introduced by subordinating conjunction or 0 (see below)
SBARQ	Direct question introduced by wh-word or wh-phrase
SINV	Declarative sentence with subject-aux inversion
SQ	Subconstituent of SBARQ excluding wh-word or wh-phrase
VP	Verb phrase
WHADVP	wh-adverb phrase
WHNP	wh-noun phrase
WHPP	wh-prepositional phrase
X	Constituent of unknown or uncertain category
Null elements	
*	"Understood" subject of infinitive or imperative
0	Zero variant of that in subordinate clauses
T	Trace–marks position where moved wh-constituent is interpreted
NIL	Marks position where preposition is interpreted in pied-piping contexts

The Stanford Parser generates a parse tree corresponding to a particular input sentence. A sample parse tree produced by the parser is shown in Fig. 3.1. The figure shows an annotated parse tree where each word or non-terminal symbol is annotated using POS tags from [16, 28]. The complete list of POS tags can be found in Table 3.1. The parse tree also shows the presence of a set of syntactic tags(bracket level tags) [16, 28]. These tags are listed in Table 3.1.

3.2 Stanford Typed Dependency Representation

The Stanford Typed Dependency Representation [5, 7] expresses grammatical relationships in terms of a directed graph [6]. A dependency or a single node in the graph can be represented as a relation triple. For example, in the independent clause *"Jack brought water""*, the relationship between the tokens "Jack" and "brought" can be represented in the form of a binary relation **"nsubj(brought-2, Jack-1)"**, which identifies *"Jack"* (word number 1 in the clause) as the subject

Table 3.2: Complete Set of Relations provided by the Stanford Typed Dependency Representation

Abbreviation	Typed Dependency
acomp	adjectival complement
advcl	adverbial clause modifier
advmod	adverbial modifier
agent	agent
amod	adjectival modifier
appos	appositional modifier
aux	auxiliary
auxpass	passive auxiliary
cc	coordination
ccomp	clausal complement
conj	conjunct
cop	copula
csubj	clausal subject
csubjpass	clausal passive subject
dep	dependent
det	determiner
discourse	discourse element
dobj	direct object
expl	expletive
goeswith	goes with
iobj	indirect object
mark	marker
mwe	multi-word expression
neg	negation modifier
nn	noun compound modifier
npadvmod	noun phrase as adverbial modifier
nsubj	nominal subject
nsubjpass	passive nominal subject
num	numeric modifier
number	element of compound number
parataxis	parataxis
pcomp	prepositional complement
pobj	object of a preposition
poss	possession modifier
possessive	possessive modifier
preconj	preconjunct
predet	predeterminer
prep	prepositional modifier
prepc	prepositional clausal modifier
prt	phrasal verb particle
punct	punctuation
quantmod	quantifier phrase modifier
ref	referent
root	root
tmod	temporal modifier
vmod	reduced non-finite verbal modifier
xcomp	open clausal complement
xsubj	controlling subject

for the verb *"brought"* (word number 2 in the sentence). The Stanford Dependency Manual lists 50 such relations, of which 48 can be seen in Table 3.2.

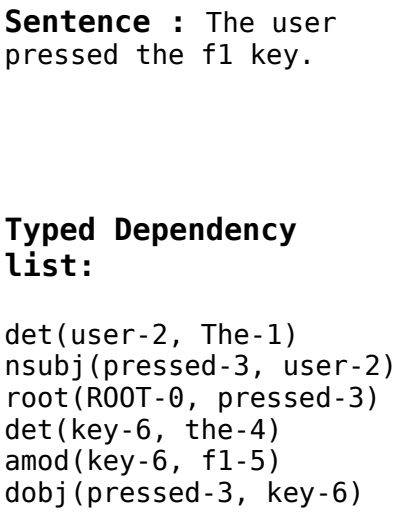


Figure 3.2: An Example Typed Dependency List

The first argument of the binary relation is referred to as the governor or head and the second argument is referred to as the dependent [6]. In this thesis we refer to a “dependency" as a triple represented by *relation(governor, dependent)*. A fully annotated typed dependency representation is shown in Fig. 3.2.

3.3 Wordnet

WordNet [18] is a database for English lexicon. It essentially classifies words into groups or *synsets* based on similar senses generated by each word. WordNet can be considered as a repository for nouns, verbs, adjectives and adverbs. It contains a total of 117,000 synsets [`http://wordnet.princeton.edu/`] and each synset is linked to other synsets to form a network of conceptually related words. WordNet glosses are essentially definitions assigned to each word and signify the meaning of the word specific to the synset it belongs to. In our work, we use these glosses to extract meanings of words, that are of importance to our approach. Wordnet is available in the electronic form [10] and can be queried using the publicly available API RiWordNet(`http://rednoise.org/rita/reference/RiWordNet.html`). RiWord-

Net provides the interface via which we can query the WordNet dictionary for glosses. A few examples of WordNet glosses obtained via RiWordNet are enumerated below:

1. attacker/NOUN: someone who attacks
2. user/NOUN: a person who makes use of a thing; someone who uses or employs something
3. server/NOUN: a person whose occupation is to serve at table (as in a restaurant)

Since in our approach we query glosses for only nouns, each word enumerated above is associated with its part-of-speech, which is *NOUN* for all the cases. Although the definition of "server" is inappropriate to the concept of computer security, it aids us in our approach.

3.4 SentiWordnet

SentiWordNet [2, 9] attaches `Positive`, `Negative` or `Neutral` sentiment to each word in terms of a score in the range $[0.0, 1.0]$. It essentially attaches these polarities to each synset of a word, i.e. each WordNet [10, 18] sense a word is assigned to. The sum of the scores assigned to a word is always 1.0 for a single synset the word belongs to. However, for this work, we use a simplified version of SentiWordNet 3.0 [`http://sentiwordnet.isti.cnr.it/`] which uses only word labels (adjective, noun, verb, adverb) to generate sentiment scores instead of synsets (all senses of a word which represent the same label, are clubbed together). Each word is associated with either a positive or a negative score, and the resultant score is calculated by subtracting the negative score from the positive score. The synsets/words under the same label are represented as *word and associated rank (eg. impulsive#4)*. The "rank" essentially rates a word based on its most common synset classifications. For example, in SentiWordNet 3.0, the word "impulsive" can be associated with 4 different WordNet synsets.

Table 3.3 is a inversely ranked order of most common appearances of the keyword "impulsive", according to WordNet 3.0. The column `diff` marks the difference between positive and negative scores for each sense of "impulsive". The final score is calculated as

Table 3.3: Sentiment Score Calculation for “Impulsive"

POS	Rank	Meaning	+ score	- score	diff	diff\rank	1\rank
a	5	characterized by undue haste and lack of thought or deliberation	0	0.625	-0.625	-0.125	0.2
a	4	determined by chance or impulse or whim rather than by necessity or reason	0	0	0	0	0.25
a	3	having the power of driving or impelling	0.25	0	0.25	.083	.333
a	2	without forethought	0.125	0	0.125	0.062	.5
a	1	proceeding from natural feeling or impulse without external stimulus	0	0	0	0	1
					Total	.02	2.28

$$\frac{\sum_i^{number\ of\ wordnet\ occurences} \frac{diff_i}{rank_i}}{\sum_i^{number\ of\ wordnet\ occurences} rank_i} \Longrightarrow \frac{0.02}{2.28} = .009$$ [6]

Thus the word “impulsive" generates a `positive` sentiment in this case.

3.5 CVE-Details

CVE-Details is a vulnerability database which provides semi-structured information on vulnerabilities identified by the CVE-identifier[7]. CVE-DETAILS reports on the same vulnerabilities that are published in the National Vulnerability Database (NVD). As a result, the vulnerability descriptions provided by CVE-details are similar to that provided by NVD. However, CVE-Details[8] also acts as a repository for software names. These are the software which have been affected by vulnerabilities having CVE-identifiers. As a result, making use of the software repository at CVE-details limits the scope of our approach to extracting only those software names which have been affected by vulnerabilities having CVE-identifiers. However, in absence of an absolute source of software names we consider CVE-Details as the best available repository.

Fig. 3.3 shows an example of a list of software names displayed by CVE-DETAILS when queried using the keyword “CSS". CVE-DETAILS supports querying the software database using HTTP GET parameters and SQL-like syntax. For example, the page shown in Fig. 3.3 is generated by querying the CVE-DETAILS database at `http://www.cvedetails.com/product-search.php` with the following parameters:

[6]formula source: sample code at `http://sentiwordnet.isti.cnr.it/`

[7]The CVE identifier is a unique identifier assigned to reported vulnerabilities by the MITRE corporation

[8]`http://www.cvedetails.com/product-search.php`

Product Search

	Product Name	Vendor Name	Number of Vulnerabilities	Product Type
1	Css Secure Content Accelerator	Cisco	3	Application
2	Css11000 Content Services Switch	Cisco	5	Application
3	Cssearch	Cgiscript.net	1	Application
4	Js Css Optimizer	Axel Jung	1	Application
5	Live Css	Guy Bedford	1	Application
6	Oscss	Oscss	2	Application
7	Php & Css Bbs	Phpspot	2	Application
8	Scssboard	Scssboard	6	Application

Figure 3.3: Software Name Repository at CVE-DETAILS

`vendor_id=0&search=%25CSS%25`

As noticed above, the parameter “search" is assigned the value “%25CSS%25" which in plain text is “%CSS%". This is similar to querying using the SQL operator “like". For the purpose of automated extraction, we query the CVE-DETAILS database at `http://www.cvedetails.com/product-search.php` with “search" parameters codified in this SQL wildcard-like form. Following that we extract the product and vendor names (as shown in Fig. 3.3) by parsing the HTML DOM tree representation of the web-page returned by CVE-DETAILS.

Chapter 4

Formulating the PAG concepts

The goal of this thesis is to be able to extract textual representations of the PAG concepts from vulnerability descriptions provided by vulnerability databases like NVD. In this chapter, we aim to provide an unsupervised rule-based approach to parsing vulnerability descriptions and obtaining instances of *Software and Versions*, *Attacker Actions*, *User Actions* and *Post-Conditions* from them. Our approach begins by pre-processing the retrieved text, where we break the text up into one or more sentences and remove unnecessary sections of a sentence, thereby preparing it for further analysis in the next steps of the algorithm. This is followed by two separate techniques/components which aim at extracting two different sets of PAG concepts. The first of these components is aimed at extracting *Software Names*, *Versions* and *Modifiers*. The second component focuses on extracting *Attacker Actions*, *User Actions* and *Post Conditions*. Both these components are shown in Fig. 4.1. Component dependencies like the Stanford Parser, the Stanford Typed Dependency representation, CVE-DETAILS, WordNet and SentiWordNet (refer to Chapter 3) are also shown in the figure. Each of the components is based on some rules composed by thoroughly observing grammatical and syntactical patterns occurring frequently in vulnerability descriptions. It is to be noted that vulnerability descriptions are also free text written in natural language. As a result, some of the rules are composed by observing grammatical and syntactical patterns in natural language text too. In course of describing the components, we state all the heuristics which are used to create the rules.

4.1 Pre-processing the Input Text

Pre-processing the input vulnerability descriptions involve three distinct steps namely, separating out sentences from the original text, removing any parts of the sentence which are not required in the final results and ensuring that all periods, but the terminating one, are removed from each sentence.

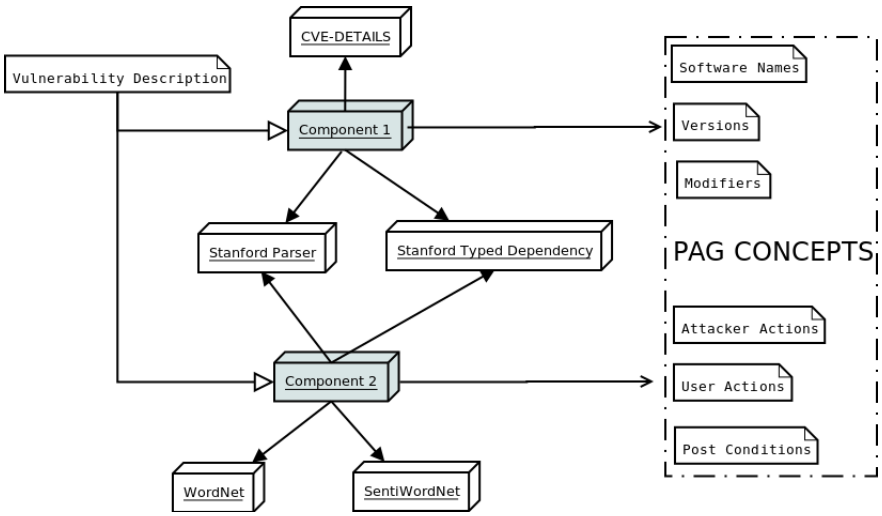


Figure 4.1: Architecture of the PAG Concept Extractor

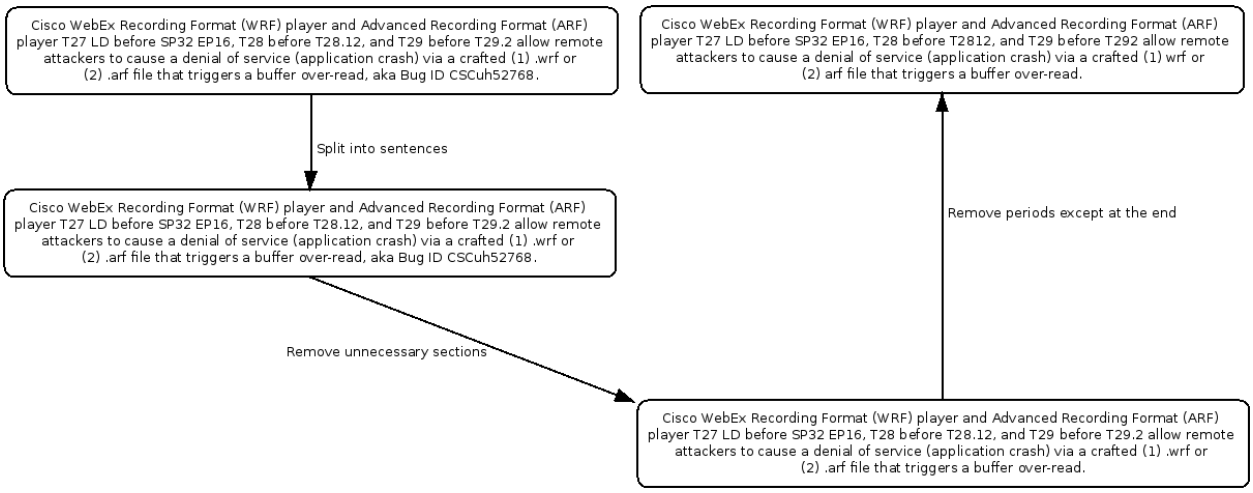


Figure 4.2: An example of vulnerability description pre-processing

Fig. 4.2 depicts a complete example of the actions taken to pre-process the input text. In the first step, the original paragraph [**source :** NVD-(*CVE-2014-2132*)] is segregated into separate sentences. In this example, we have a single sentence and hence the split operation does not have any effect. To split a paragraph into sentences we use the sentence-splitter offered by the Cognitive Computation Group at University of Illinois, Urbana-Champaign [9]. Following that, unnecessary sections are removed from the sentences. These are the sections of the input text which do not contribute to the final PAG concepts. In this case, the section “aka Bug ID CSCuh52768" is

[9] `http://cogcomp.cs.illinois.edu/software/doc/LBJ2/library/LBJ2/nlp/SentenceSplitter.html`

deemed unnecessary and hence removed. Enumerating all possible unnecessary sections for all vulnerability databases is beyond the scope of this thesis and hence we manually identified a few such sections which are repeated frequently in NVD descriptions and removed then automatically. Finally, it is ensured that periods within a sentence are removed and the sentence ends with a period. In this example, terms like “T28.12" and “T29.2" are reduced to “T2812" and “T292" respectively. Removing inter-sentence periods ensures that additional natural language processing does not treat a single sentence as a sequence of sentences. For this same purpose, it is also ensured that input sentence ends with a period.

4.2 Extracting Software Names, Versions and Modifiers (Component 1)

The approach to extracting vulnerable *Software Names* and their related *Versions* and *Modifiers* relies on identifying words in the input vulnerability description which bear some resemblance to common software names. The resemblance is quantified in terms of probabilities and calculated using a set of algorithms which are described in this section. The flow of logic used in this approach is shown in Fig. 4.3.

The process of extracting *Software Names* and their related *Versions* and *Modifiers* begins by creating a *Sentence-Bundle*. A sentence-bundle is essentially a collection of groups of consecutive proper nouns in a sentence. A sentence-bundle is formally defined in section 4.2.1. Software names matching to the proper nouns are then downloaded from CVE-DETAILS (see section 3.5). Groups of proper nouns are then matched to the downloaded software names to identify instances of *Software Names* in the input sentence (vulnerability description). Finally, a clean-up procedure is performed where duplicate or irrelevant instances of *Software Names* are removed. This is followed by identifying instances of *Versions*[10] in the input sentence. *Versions* cannot be treated as independent entities. In other words, an instance of a *Version* is always linked to some instance

[10]Software updates and editions are also treated as versions in this thesis

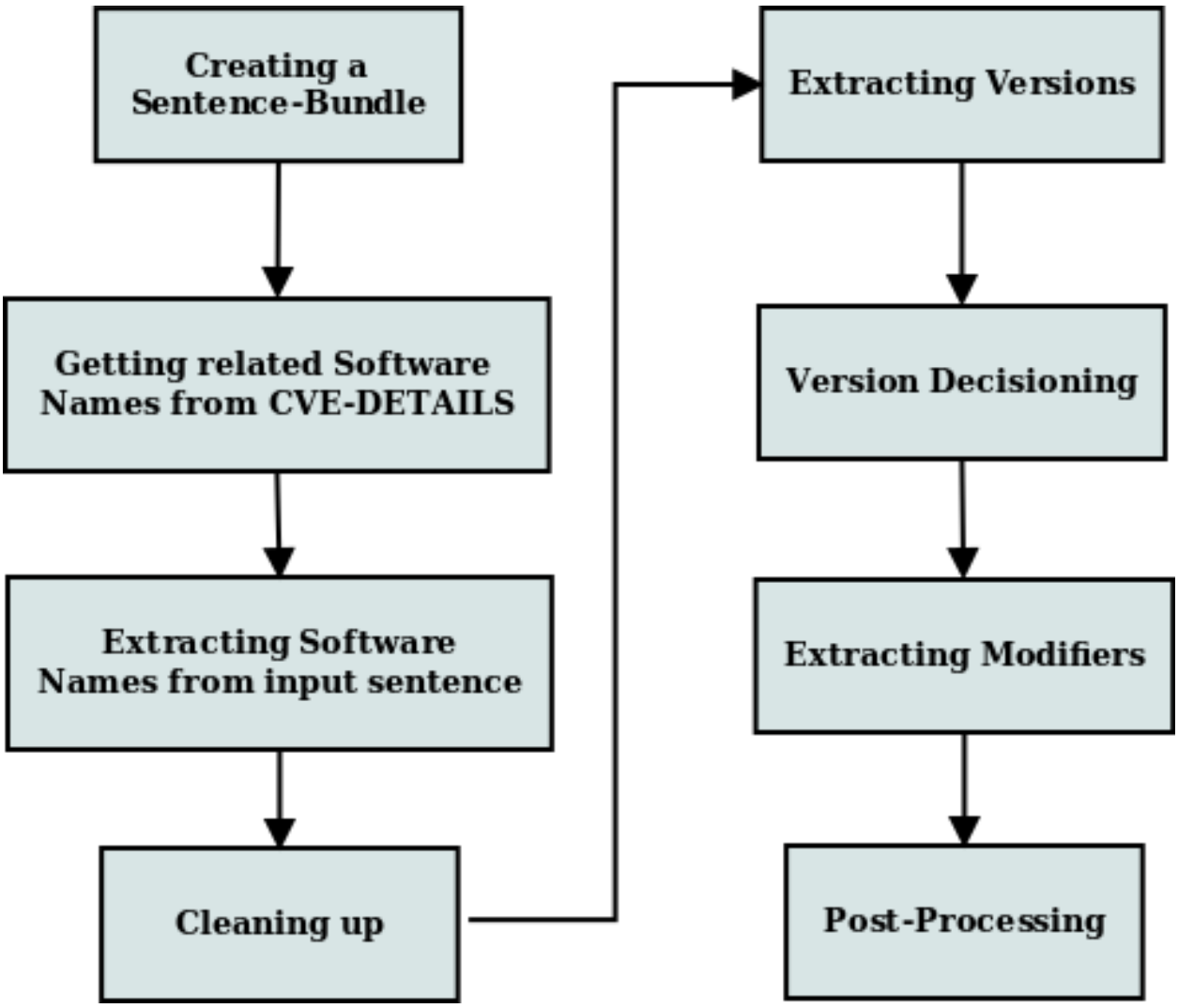


Figure 4.3: Component 1 Workflow Diagram

of a *Software Name*. Thus once the versions are identified they are then linked to their corresponding *Software Names*. Instances of *Modifiers* are then extracted. Modifiers are dependent on versions since they essentially quantify ranges of versions. As a result, instances of *Modifiers* are also linked to their corresponding versions. Eventually, a post-processing procedure is carried out where similar software-version-modifier triples are combined.

To demonstrate the flow of the algorithms presented in this section we make use of the sanitized (section 4.1) version of the vulnerability description mentioned below. This description is taken from the National Vulnerability Database (CVE-2014-7298). We chose this description as it allows us to demonstrate all facets of our algorithms.

"*adsetgroups in Centrify Server Suite 2008 through 20141 and Centrify DirectControl 3x through 420 on Linux and UNIX allows local users to read arbitrary files with root privileges by leveraging improperly protected setuid functionality.*"

4.2.1 Creating a Sentence-Bundle

Heuristic 1. *Software names are generally expressed as groups of consecutive proper nouns.*

- *Eg. "**Microsoft Windows** and **Adobe Flash Player**"*

Heuristic 2. *If an independent clause or sentence is split into two halves around a constituent verb, software names are included in the first or left half if the total number of proper nouns (Software Names), numerical terms (Versions) and prepositions (Modifiers) is higher for the left half. For an independent clause, the first or left half is a noun phrase which contains the subject of the clause.*

- *Eg. When the independent clause "**Microsoft Windows** 7 through 10 is vulnerable to a buffer overflow attack." is split around the verb "is", the software term **Microsoft Windows** appears on the left or first half since the total number of proper nouns [2] ("Microsoft", "Windows"), numerical terms [2] ("7","10") and prepositions [1] ("through") is higher for the left half. Note that **Microsoft Windows** is also the subject of the clause.*

The task of extracting *Software Name*s, *Versions* and *Modifiers* begins by creating a *sentence-bundle*.

Definition (Sentence-Bundle)**.** *A sentence-bundle is a structured representation of a sentence or a part of a sentence and is used for the purpose of extracting Software Names. A sentence-bundle can be viewed as a collection of groups of consecutive proper nouns (NNP) within a sentence or a part of the sentence. A sentence-bundle also contains the Stanford Typed Dependency representation of the sentence.*

The process of preparing a sentence-bundle is described in Algorithm 1. An example is also shown in Fig. 4.4.

The algorithm takes in a sanitized version of the input text and the parts-of-speech tagged tree generated by the Stanford Parser (described in Chapter 3). According to heuristic 1 and 2 software

Algorithm 1 $CreateSentenceBundle$

Require: Stanford POS tagged tree (Tree), Sentence

 # Get resultantHalf #
$resultantHalf \leftarrow \{\}$
$resultantDependencyList \leftarrow \{\}$
$verbToStop \leftarrow$ first verb from the start of the sentence
while (End of sentence is not reached) **do**
  $firstHalf \leftarrow \{word | word \in$ All words from the start of the sentence till $verbToStop\}$
  $secondHalf \leftarrow \{word | word \in$ All words from $verbToStop$ till the end of the sentence$\}$
  $score1 = \frac{|\{x|x \in 'CD', 'NNP', 'IN' \text{ tagged words in } firstHalf\}|}{|\{x|x \in \text{All words in } firstHalf\}|}$
  $score2 = \frac{|\{x|x \in 'CD', 'NNP', 'IN' \text{ tagged words in } secondHalf\}|}{|\{x|x \in \text{Allwords in } secondHalf\}|}$
  if $(score1 > score2)$ **then**
    $resultantHalf \leftarrow firstHalf$
    $resultantDependencyList \leftarrow$ Stanford Typed Dependency Tree for $firstHalf$
    EndLoop
  else
    $verbToStop \leftarrow$ next verb from $verbToStop$
  end if
end while
 # Get NNPgroups #
$NNPgroups \leftarrow \{\}$
$NNPgroup \leftarrow \{\}$
for all $(word \in resultantHalf)$ **do**
  $NNPgroup \leftarrow (NNPgroup \cup word)$ if POS-tag = 'NNP'
  if (POS-tag $\notin \{'NNP', 'CD'\}$) **then**
    $NNPgroups \leftarrow NNPgroups \cup NNPgroup$
    $NNPgroup \leftarrow \{\}$
  end if
end for
$sentence\text{-}bundle \leftarrow \langle NNPgroups, resultantHalf, resultantDependencyList \rangle$
return $sentence\text{-}bundle$

names are often expressed as proper noun subjects in a sentence. However, their might be other proper nouns in the object part of the sentence. Thus to select the proper half of the sentence we split the sentence into two halves, centered around the first found verb. Both halves are then attributed to a probability of being selected. The probability is calculated as:

$$\frac{|\{x|x \in' CD',\ 'NNP',\ 'IN' \text{ tagged words in } half\}|}{|\{x|x \in \text{All words in } half\}|}$$

Essentially, the probability denotes the fraction of 'CD' (Cardinal Numbers), 'NNP' (Proper Nouns) and 'IN' (Prepositions) labeled words over the total number of words in that half. The choice of the above mentioned three tags is guided by the fact that version numbers are generally tagged as 'CD', while software names are tagged as 'NNP' and modifiers like "through" are tagged as 'IN'. If the first half attains a higher score than the second half, the former is chosen as the resultant half or the half which contains the *Software Name*s. On the other hand, if the second half leads, the next found verb in the sentence is chosen and the above-described procedure is repeated. In Fig. 4.4 the first half "*adsetgroups in ... and Unix*" has a total of 14 'CD', 'NNP' and 'IN' tagged words, compared to only 2 such words for the second half. As a result, it is chosen as the resultant half based on its precedence in terms of the score. For simplicity, we refer to the resultant half as a sentence.

Next, following heuristic 1 consecutive proper nouns in the resultant half are clubbed together. These groups are referred as to *NNPgroup*s and are defined below.

Definition (NNPgroup, NNP, Query)**.** *An NNPgroup is a collection of consecutive proper nouns in a block of text, usually the subject half of a sentence. The proper nouns constituting an NNPgroup are considered to be consecutive if they either occur in sequence or with interleaved cardinal numbers (CD). Each component proper noun of an NNPgroup is referred to as an NNP. An ordered collection of proper nouns from the same NNPgroup is referred to as a Query.*

NNPgroups obtained from the input vulnerability description are shown in Fig. 4.4 as a part of the sentence-bundle. A Stanford Typed Dependency list of the resultant half is also added to the sentence-bundle.

4.2.2 Getting Software Names from CVE-DETAILS

Heuristic 3. *Majority of software names are present in the CVE-DETAILS software repository.*

- *Eg.* ***Windows*** *is present in the CVE-DETAILS repository under the vendor* ***Microsoft****.*

Algorithm 2 takes a sentence-bundle as an input and generates a set of product-vendor pairs where every the product matches to any at least one of the NNPs in the sentence. Fig. 4.5

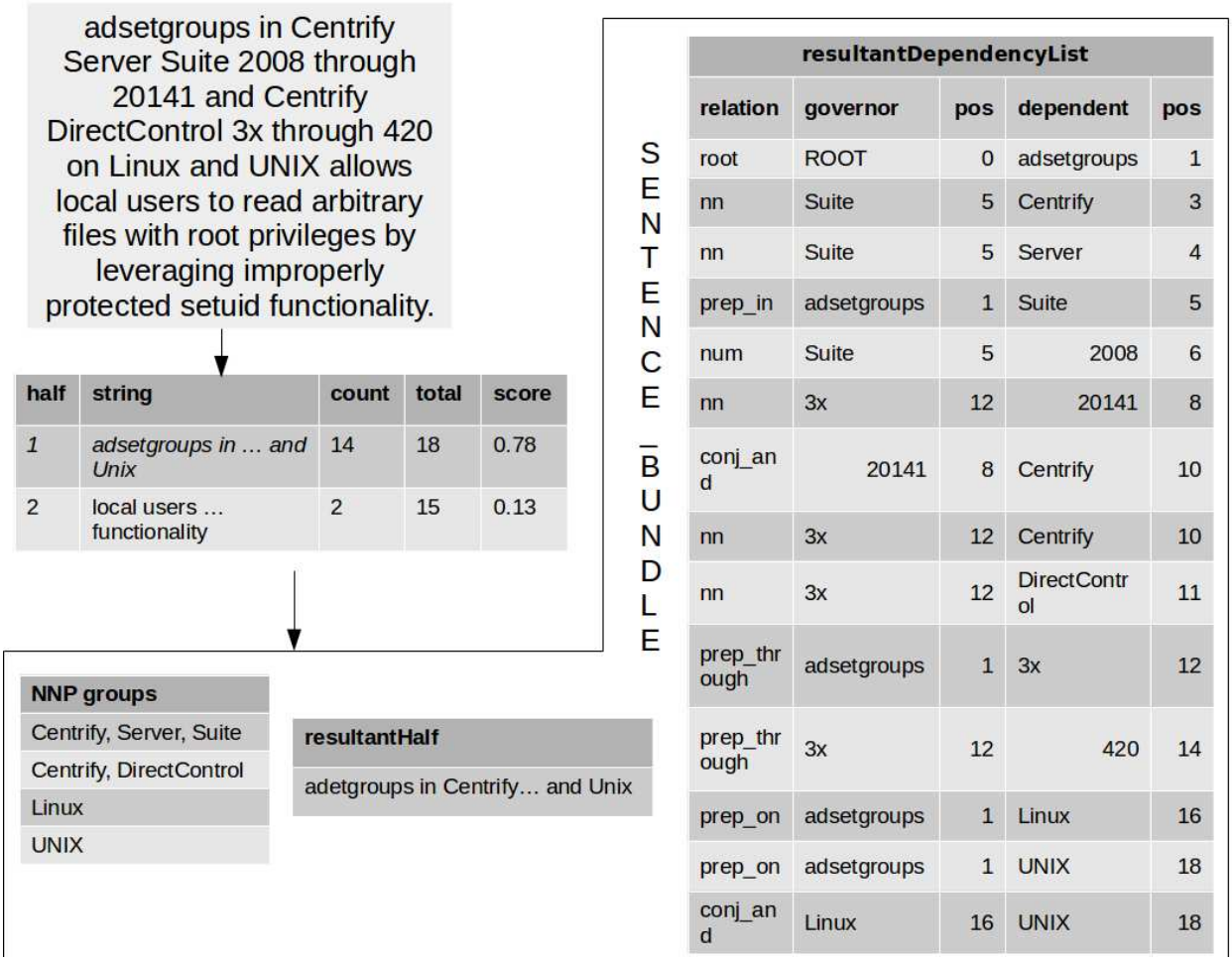


Figure 4.4: Forming a Sentence-Bundle Tuple

shows an example of the product-vendor pairs extracted from CVE-DETAILS. CVE-DETAILS provides a structured repository for common CVE-vulnerability affected software. As mentioned in Chapter 3 CVE-DETAILS can be queried at a base URL (`http://www.cvedetails.com/product-search.php`) using an SQL-like wild-card (%) operator. We send four different queries to CVE-DETAILS:

- CVE-DETAILS base URL + NNP
- CVE-DETAILS base URL + NNP%
- CVE-DETAILS base URL + %NNP
- CVE-DETAILS base URL + %NNP%

Query	Product	Vendor
http://www.cvedetails.com/product-search.php?vendor_id=0&search=**Centrify**	No Results	No Results
http://www.cvedetails.com/product-search.php?vendor_id=0&search=**Centrify+%25**	Centrify Deployment Manager	Centrify
	Centrify Suite	Centrify
http://www.cvedetails.com/product-search.php?vendor_id=0&search=**%25+Centrify**	No Results	No Results
http://www.cvedetails.com/product-search.php?vendor_id=0&search=**%25+Centrify+%25**	No Results	No Results

Figure 4.5: Results from CVE-DETAILS

Each of these four queries return 0 or more product-vendor pairs from CVE-DETAILS. These product-vendor pairs are scrapped off the returned CVE-DETAILS product web-page. Some of the product-vendor pairs are shown in Fig. 4.5. Eventually, each product-vendor pair is assigned to the NNP (*NNP.probableSoftwareList*) which was used to query CVE-DETAILS for it.

4.2.3 Identifying Software Names in Text

Heuristic 4. *An ordered collection of proper nouns from an NNPgroup (query) can be considered as a probable software name (after matching to product-vendor pairs downloaded from CVE-DETAILS) if any of the following happen in order of precedence:*

1. *Downloaded vendor name is present in the query.*

Algorithm 2 $getSoftwareFromCVE\text{-}DETAILS()$

Require: sentence-bundle
$CVE\text{-}DETAILS\_base\_URL \leftarrow$ http://www.cvedetails.com/product-search.php?vendor_id=0&search="
for all ($NNPgroup\ in\ sentence\text{-}bundle.NNPgroups$) **do**
  for all ($NNP \in NNPgroup$) **do**
    Create **CVE-DETAILS** queries like:
    $CVE\text{-}DETAILS\_base\_URL + \{NNP, NNP\%, \%NNP, \%NNP\%\}$
    for each query **do**
      Get all possible probable product-vendor pairs.
      $NNP.probableSoftwareList \leftarrow$ all possible product-vendor pairs
    end for
  end for
end for

- *Eg. Vendor **Microsoft** is present in the query "**Microsoft Windows**"*

2. *Downloaded vendor name is present in the input sentence.*

   - *Eg. Vendor **Microsoft** is present in "**Microsoft Windows** is vulnerable to..."*

3. *Greatest score, denoting the fraction of number of words present in the query to that of the product-name, is* ≥ 0.5.

   - *Eg. Examples are provided in the algorithm description.*

Once the product-vendor pairs have been retrieved, the proper nouns in an NNPgroup are assembled together in a systematic manner to form **queries**, which are then evaluated against the retrieved product names. Algorithm 3 and Fig. 4.6 are used to demonstrate this scenario. At first, the proper nouns are assembled in order to form queries. This can be seen in Fig. 4.6 where the NNPs "Centrify", "Server" and "Suite" are assembled to form the following queries :

- "Centrify"
- "Centrify Server"
- "Centrify Server Suite"
- "Server"

Algorithm 3 $identifySoftwareNamesInText()$

Require: sentence-bundle, sentence

$Probable\text{-}Software\text{-}List \leftarrow \{\}$
for ($NNPgroup\ in\ sentence\text{-}bundle.NNPgroups$) **do**
  for ($i = 1\ to\ |NNPgroup|$) **do**
    $query \leftarrow \{\}$
    for ($j = i\ to\ |NNPgroup|$) **do**
      $query \leftarrow query \cup j^{th}\ NNP\ of\ NNPgroup$
      $productVector \leftarrow \{\}$
      for ($NNP \in query$) **do**
        for all ($product\text{-}vendor\text{-}pair \in NNP.probableSoftwareList$) **do**
          if ($product\ \supseteq\ query\ in\ same\ order$) **then**
            $productVector \leftarrow productVector \cup product\text{-}vendor\text{-}pair$
          end if
        end for
      end for
      $probWithVendorSelf \leftarrow 0.0$
      $probWithoutVendorSelf \leftarrow 0.0$
      $probNormal \leftarrow 0.0$
      $queryWithVendorSelf \leftarrow \emptyset$
      $queryWithoutVendorSelf \leftarrow \emptyset$
      $queryNormal \leftarrow \emptyset$
      for all ($product\text{-}vendor\text{-}pair \in productVector$) **do**
Define: $prob \leftarrow \frac{|query|}{|\{word|word \in product\}|}$
        if ($vendor \subseteq\ query$) **then**
          if ($prob > probWithVendorSelf$) **then**
            $probWithVendorSelf \leftarrow prob$
            $queryWithVendorSelf \leftarrow query$
          end if
        else if ($vendor \subseteq\ sentence$) **then**
          if ($prob > probWithoutVendorSelf$) **then**
            $probWithoutVendorSelf \leftarrow prob$
            $queryWithoutVendorSelf \leftarrow query$
          end if
        else
          if ($prob > probNormal$) **then**
            $probNormal \leftarrow prob$
            $queryNormal \leftarrow query$
          end if
        end if
      end for
      if ($queryWithoutVendorSelf \not\leftarrow \emptyset$) **then**
        Probable-Software-List $\leftarrow$ *Probable-Software-List* $\cup$ *queryWithoutVendorSelf*

Algorithm 3 $identifySoftwareNamesInText()$ *continued..*

```
            else if (queryWithVendorSelf ⊄ ∅) then
                Probable-Software-List ← Probable-Software-List ∪ queryWithVendorSelf
            else
                if (probNormal ≥ .5) then
                    Probable-Software-List ← Probable-Software-List ∪ queryNormal
                end if
            end if
        end for
    end for
end for
return Probable-Software-List
```

- "Server Suite"
- "Suite"

All product names retrieved previously by querying CVE-DETAILS with the NNPs in a query are scanned to see if they contain the query elements in their specified order. If not, those product names are rejected and the selected ones are put into a list. As an example, in Fig. 4.6, query "Server Suite" is evaluated against all product names retrieved by querying CVE-DETAILS against query elements "Server" and "Suite". Of these, two examples are shown where the product "Server Protection Suite", along with its associated vendor (not shown in the Fig. 4.6), is kept in the final list as it contains the words "Server" and "Suite" from the original query in the same order. However, the product "Server" is rejected since it does not contain the word "Suite".

Once the above-mentioned list is created, each product-vendor pair in the list is compared to the input query to generate a selection score. The score is defined as:

$$\frac{|query|}{|\{word|word \in product\}|}$$

Here *product* refers to the product names. Thus the score denotes the fraction of the size of the query over the total number of words in the matching product name. The product with the highest score is selected as the closest match to the query. However, if the name of the vendor for

NNPGroup: Centrify, Server, Suite

Query	Product	Selected
Centrify	Centrify Deployment Manager	True
Centrify Server	Centrify Deployment Manager	False
Centrify Server Suite	Centrify Deployment Manager	False
Server	**Server**	True
Server Suite	**Server Protection Suite**	True
	Server	False
Suite	Suite	True

Query	Product	Vendor	Status	Score
Centrify	Centrify Deployment Manager	Centrify	queryWithVendorSelf	0.33
	Centrify Suite	**Centrify**	**queryWithVendorSelf**	**0.5**
Centrify Server	No Selection	No Selection	N/A	N/A
Centrify Server Suite	No Selection	No Selection	N/A	N/A
Server	**Server**	**Vmware**	**queryNormal**	**1.0**
	Server And Application Monitor	Solarwinds	queryNormal	0.25
Server Suite	**Server Protection Suite**	**CA**	**queryNormal**	**0.67**
	Client Server Messaging Suite	Trend Micro	queryNormal	0.5
Suite	Suite	Movavi	queryNormal	1.0
	Centrify Suite	**Centrify**	**queryWithoutVendorSelf**	**0.5**

Figure 4.6: Selection of Software

a product appears in the original sentence or in the query, then that product receives precedence. This can be seen in Fig. 4.6 where the query “Suite" is assigned to the product name “Centrify Suite" (highlighted in bold) (as the corresponding vendor “Centrify" appears in the original text), in-spite of the product “Suite" having a greater score. The query “Centrify" has both vendor names which match to the query, but the one having a higher score (highlighted in bold) is selected. Similarly, the query “Server Suite" generates two product names (“Server Protection Suite" and “Client Server Messaging Suite") by virtue of having scores greater than .5. Out of these, the former (highlighted in bold) is selected, due to its higher score. The queries “Centrify Server" and “Centrify Server Suite" are rejected since they do not yield any matching software names.

Algorithm 3 returns a set of queries which have been identified as probable software names. We refer to the set as *Probable-Software-List*.

4.2.4 Cleaning up

Once the software names have been identified, a clean-up procedure is run which removes any redundant entries by performing the following steps:

1. If a software name query is a substring of another software name query in the same NNPgroup, then the substring software is removed. For example, from Fig. 4.6 queries/software names “Server" and “Suite" are removed as they are substrings of the query “Server Suite".

2. If the vendors are not mentioned in the original text, they are not considered as vendors. The significance of this step is enhanced when heuristics 5 is considered in the next subsection. According to that heuristic, if vendor names are rejected they can become potential version/edition candidates. For example, from Fig. 4.6 vendor name “CA" is not considered as an eventual vendor name for query “Server Suite" since it does not appear in the original input sentence.

3. If the total score of software queries in an NNPgroup is less than or equal to .5 then that NNPgroup is rejected. In this example, all NNPgroups have a total score of above .5 and hence are not rejected.

4.2.5 Identifying Versions

Heuristic 5. *In a sentence, terms starting with an integer ASCII character can be treated as versions.*

• *Eg. In “Microsoft Windows **7x**." “7x" can be as a version.*

In a sentence, proper noun modifiers of software names, which are not vendor names, can be treated as updates and editions (versions).

• *Eg. In “Microsoft Windows **SP2**." “SP2", being a proper noun modifier of “Windows", can be as an update or edition.*

Algorithm 4 $identifyVersions()$

Require: Probable-Software-List, sentence-bundle

procedure ADDADJUNCTVERSION(relationList, target, step, depth, possibleAdjuncts, possibleVersions)
    $queryAdjunctList \leftarrow \{\}$
    for ($dependency \in sentence\text{-}bundle.resultantDependencyList$) **do** *# dependency = $\langle relation, governor, dependent\rangle$ #*
        if ($relation \in relationList$) **then**
            if ($dependent \in target$) **then**
                if ($governor \notin any\ query\ in\ Probable\text{-}Software\text{-}List\}$) **then**
                    $value = governor$
                    if ($step = 1 \vee step = 3$) **then**
                        $distance = (governor.pos - dependent.pos)$ *# pos = starting position from beginning of the sentence #*
                    else if ($step = 2 \vee step = 4$) **then**
                        $distance = (governor.pos - dependent.pos) + distance$ of matching target element
                    end if
                end if
            else if ($governor \in target$) **then**
                if ($dependent \notin any\ query\ in\ Probable\text{-}Software\text{-}List\}$) **then**
                    $value = dependent$
                    if ($step = 1 \vee step = 3$) **then**
                        $distance = (dependent.pos - governor.pos)$
                    else if ($step = 2 \vee step4$) **then**
                        $distance = (dependent.pos - governor.pos) + distance$ of matching target element
                    end if
                end if
            end if
            $depth = depth$
            $tup \leftarrow \langle value, distance, depth, relation\rangle$
            if ($first\ character\ of\ value \in \mathbb{Z}$) **then**
                $possibleVersions \leftarrow possibleVersions \cup tup$
            else
                if ($step = 1 \vee step = 2$) **then**
                    $possibleAdjuncts \leftarrow possibleAdjuncts \cup tup$
                end if
            end if
        end if
    end for
end procedure
$depth = 1$

Algorithm 4 $identifyVersions()$ *continued..*

for ($query \in Probable\text{-}Software\text{-}List$) **do**
  $possibleAdjuncts \leftarrow \{\}, possibleVersions \leftarrow \{\}$ *# sets of 4-tuples #*
  *# * = wildcard character #*
  # pA = possibleAdjuncts, pV = possibleVersions #
  ADDADJUNCTVERSION($\{'nn','prep\_*','dep'\}, query, 1, depth, pA, pV$) *# step 1 #*
  while true **do**
    $depth = depth + 1$
    $prevAd \leftarrow possibleAdjuncts$
    ADDADJUNCTVERSION($\{'nn','prep\_*','dep','conj'\}, pA, 2, depth, pA, pV$) *# step 2 #*
    if ($|prevAd| = |possibleAdjuncts|$) **then end loop**
    end if
  end while
  $depth = 1$
  ADDADJUNCTVERSION($\{x|x \in all\ possible\ relations\}, query, 3, depth, pA, pV$) *# step 3 #*
  while true **do**
    $depth = depth + 1$
    $prevVers \leftarrow possibleVersions$
    ADDADJUNCTVERSION($\{x|x \in all\ possible\ relations\}, pV, 4, depth, pA, pV$) *# step 4 #*
    if ($|prevVers| = |possibleVersions|$) **then end loop**
    end if
  end while
  $queryAdjunctList \leftarrow queryAdjunctList \cup \langle query, possibleAdjuncts, possibleVersions \rangle$
end for
 # Identify Updates and Editions #
for ($queryAdjunct \in queryAdjunctList$) **do**
  for ($adjunct \in queryAdjunct.possibleAdjuncts$) **do**
    if (($adjunct$ is NNP-tagged) $\wedge$ ($adjunct.relation =' nn'$) $\wedge$ ($adjunct$ is not a vendor)) **then**
      $queryAdjunct.possibleVersions \leftarrow queryAdjunct.possibleVersions \cup adjunct$
    end if
  end for
end for

Heuristic 6. *Version terms can have direct or indirect grammatical dependencies on probable software names. In the case of indirect relations the intermediate grammatical dependencies are limited to the following list of relations:*

- *Noun Compound Modifiers (nn)*

- *Prepositions (prep)*
- *Conjunctions (conj)*
- *Unknown Dependencies (When the grammatical dependency cannot be determined in the worst case)*

• *Eg. In sentences "Windows 7 is vulnerable." and "Windows versions 7 is vulnerable." the following dependencies can be found.*

```
a> Windows <---- 7    ......
b> Windows <---- version <----- 7    ....
```

In the first sentence version "7" is directly dependent on software name "Windows" and in the second sentence it is indirectly dependent via term "version".

Going by heuristics 6, in order to identify versions we need to traverse the Stanford Typed Dependency tree and add all typed dependencies directly or indirectly related to the each NNP in a query. While direct relations are obvious, we formalize the concept of indirect relations by introducing the concept of *adjuncts*.

Definition (Adjunct)**.** *An adjunct is a term in a sentence that has a direct or indirect grammatical dependency on a software term. An adjunct can grammatically link a software term and a version term, a software term and another adjunct and two different adjuncts. By heuristic 6 the chain of grammatical dependencies between an adjunct and a software terms can be a combination of the following relations:*

- *Noun Modifiers (nn)*
- *Prepositions (prep)*
- *Conjunctions (conj)*

Step 1

Query	Versions	Version Distance	Version Pos	Version Depth	Adjunct	Adjunct Distance	Adjunct Pos	Adjunct Depth	Adjunct Tag
Centrify									
Server Suite					adsetgroups	-4	1	1	prep_in
Centrify	3x	2	12	1					
DirectControl	3x	1	12	1					
Linux					adsetgroups	-15	1	1	prep_on
UNIX					adsetgroups	-17	1	1	prep_on

Step 2

Query	Versions	Version Distance	Version Pos	Version Depth	Adjunct	Adjunct Distance	Adjunct Pos	Adjunct Depth	Adjunct Tag
Centrify									
Server Suite	3x	7	12	2	adsetgroups	-4	1	1	prep_in
Centrify	3x	2	12	1					
DirectControl	3x	1	12	1					
Linux	3x	-4	12	2	adsetgroups	-15	1	1	prep_on
UNIX	3x	-6	12	2	adsetgroups	-17	1	1	prep_on

Step 3

Query	Versions	Version Distance	Version Pos	Version Depth	Adjunct	Adjunct Distance	Adjunct Pos	Adjunct Depth	Adjunct Tag
Centrify									
Server Suite	3x,2008	7,1	12,6	2,1	adsetgroups	-4	1	1	prep_in
Centrify	3x, 20141	2,-2	12,8	1,1					
DirectControl	3x	1	12	1					
Linux	3x	-4	12	2	adsetgroups	-15	1	1	prep_on
UNIX	3x	-6	12	2	adsetgroups	-17	1	1	prep_on

Step 4

Query	Versions	Version Distance	Version Pos	Version Depth	Adjunct	Adjunct Distance	Adjunct Pos	Adjunct Depth	Adjunct Tag
Centrify									
Server Suite	3x,2008,20141,420	7,1,3,9	12,6,8,14	2,1,3,3	adsetgroups	-4	1	1	prep_in
Centrify	3x,20141,420	2,-2,4	12,8,14	1,1,2					
DirectControl	3x,20141,420	1,-3,3	12,8,14	1,2,2					
Linux	3x,20141,420	-4,-8,-2	12,8,14	2,3,3	adsetgroups	-15	1	1	prep_on
UNIX	3x,20141,420	-6,-10,-14	12,8,14	2,3,3	adsetgroups	-17	1	1	prep_on

Figure 4.7: Identifying Versions

- *Unknown Dependencies (When the grammatical dependency cannot be determined in the worst case)*

Thus, the term “version" used in the example for heuristics 6 is essentially an adjunct linking the software term and the version term. It is to be noted that version terms can also be linked to software terms via other version terms.

Algorithm 4 and Fig.4.7 demonstrate the process of identifying versions. Algorithm 4 describes a procedure *addAdjunctVersion* which adds a version or an adjunct to the list of versions/adjuncts attributed to a query (probable software name). It takes in six arguments. The role of each of these six arguments is mentioned below.

1. **relationList**: The set of relations to be matched. The set of relations vary at different calls of the procedure. While identifying adjuncts and version (step 2 and 4) the set of relations is limited to subsets of the relations mentioned in heuristic 6. While adding only versions it is a universal set of all relations supported by the Stanford Typed Dependency representation (refer to Table 3.2).

2. **target**: In process of finding adjuncts or versions, the procedure *addAdjunctVersion* performs two string matches: matching relations as described in the previous bullet and matching a term of the dependency (governor or dependent). If the governor matches, the dependent is considered as a possible adjunct/version and vice-versa. The target set contains the terms to be matched to the governor or dependent of a relation. For example, if the target set is a singleton like "Microsoft", it would match to dependencies like nn("Microsoft","version") where the governor matches and the dependent can be considered as a possible adjunct.

3. **step**: The step helps in distinguishing different calls to the procedure. Since different calls use different instances of arguments, the step variable helps the procedure to function differently for different calls.

4. **depth**: The depth argument denotes the number of branches in the Stanford Typed Dependency tree between the newly added adjunct or version dependency and the dependency which acts as the root dependency. The root dependency is essentially the first matching dependency added at steps 1 and 3. At these steps, the depth is 1. For the other steps (2 and 4)

it is incrementally increased. As an example, in Fig. 4.7 version “3x" is directly dependent on query “Centrify" (refer to *resultantDependencyList* from Fig. 4.4). This results in a depth of 1 for the term “3x". If another relation is dependent on version “3x", its depth will be 2 since it is 2 steps away from a direct relation to the query element “Centrify".

5. **possibleAdjuncts**: The current set of adjuncts associated with a query. Further adjuncts are added to this list.

6. **possibleVersions**: The current set of versions associated with a query. Further versions are added to this list.

Essentially the procedure *addAdjunctVersion* scans all dependencies from the Stanford Typed Dependency tree (Sentence-Bundle.resultantDependecyList). If the governor (or dependent) matches an element from the target set and corresponding relation matches to the relationList it creates an adjunct tuple by adding the dependent (or governor), the distance and the depth of the dependent. *Distance* measures the distance of the adjunct from the query element. For example, in Fig. 4.7 query element “Centrify" is the 10^{th} word and version “3x" is the 12^{th} word of the sentence, which results in a distance of $12 - 10 = 2$. If a second word is related to version “3x" its distance would be calculated as the sum of the distance of “3x" (2) and the distance of the newly related word from “3x". The procedure is called 4 times (steps 1, 2, 3 and 4 in) for each query in the list of probable software names. After the four calls each query is attributed to a list of *possibleAdjuncts* and *possibleVersions* in line 55 of Algorithm 4.

Fig. 4.7 demonstrates the four calls made to the “addAdjunctVersion" procedure at steps 1, 2, 3 and 4 of Algorithm 4. In step 1, our approach selects all dependencies matching the query elements and having relation names ’nn’, ’prep_*’ (* is a wildcard here) and ’dep’. For example, from Fig. 4.4 (resultantDependencyList) term “3x" matches to query element “Centrify" (its relation is ’nn’) and since it starts with an Integer (3), it is termed as a *version*. The term “adsetgroups" matches with query element “Linux" but since it does not start with an integer it is added as an *adjunct*. In step 2, all dependencies which are directly or indirectly related to the currently added adjuncts

and have relations in 'nn', 'prep_*', 'dep' or 'conj' are considered. From Fig. 4.4 version "3x" is directly related to term "adsetsgroups" and hence is indirectly related to query "Server Suite" at a depth of 2 and distance of 7. In step 3, all dependencies are matched against query elements and are added as versions if they match and satisfy the criteria to be treated as versions. From Fig. 4.4 dependent "2008" is directly related to query element "Suite" and starts with an integer ("2"). It is thus added as a version. In step 4 all relations which directly or indirectly match currently added versions are added as versions. This leads to version "420" being added to the version list of query "Server Suite", since it is related to version "3x" (refer to Fig. 4.4). Finally in lines 57 to 63 of Algorithm 4, as stated in heuristic 5, adjuncts which are proper noun modifiers are added as versions if they do not match previously (section 4.2.2) identified vendor names. Since the adjunct "adsetsgroups" is neither a proper noun nor a noun modifier, it is not added as an update or edition. Eventually the queries are attributed to a list of *possibleVersions* and added to the list *queryAdjunctList*.

4.2.6 Version Decisioning

Heuristic 7. *In a single sentence vulnerability description, versions are aligned in the following manner:*

1. *When read from left, if the first and last occurrences of version numbers precede their corresponding software names then all other version names follow the same alignment i.e. precede their corresponding software names.*

   • *Eg. Microsoft Windows **7**, Adobe Acrobat Reader **9.0** and Mozilla Firefox **24** are vulnerable.*

2. *When read from left, if the first and last occurrences of version numbers succeed their corresponding software names then all other version names follow the same alignment i.e. succeed their corresponding software names.*

   • *Eg. Version **7** of Microsoft Windows, **9.0** of Adobe Acrobat Reader and **24** of Mozilla Firefox are vulnerable.*

Query	Versions	Version Distance	Version Depth	Prob-Score
Centrify				
Server Suite	3x 2008 20141 420	7 1 3 9	2 1 3 3	.071 1 .11 ..04
Centrify	3x 20141 420	2 2 4	1 1 2	.5 -.5 .125
DirectControl	3x 20141 420	1 -3 3	1 2 2	1 -.17 .17
Linux	3x 20141 420	-4 -8 -2	2 3 3	-.125 -.041 -.17
UNIX	3x 20141 420	-6 -10 -14	2 3 3	-.083 -.03 -.02

Query	Versions
Centrify	
Server Suite	2008 20141
Centrify	
DirectControl	3x 420
Linux	
UNIX	

Figure 4.8: Decisioning in Assigning Versions

3. *When read from left, if the first and last occurrences of version numbers precede and succeed their corresponding software names respectively then all other version names can either precede or succeed their corresponding software names.*

   • *Eg. Version **7** of Microsoft Windows, Adobe Acrobat Reader **9.0** and Mozilla Firefox **24** are vulnerable.*

Heuristic 8. *Versions are most closely related to their software name counter-parts and this relation can be codified empirically.*

• *Eg. In the sentence, "Microsoft Windows 7 and Mozilla Firefox 24 are vulnerable." both the terms "Windows" and "Firefox" have grammatical dependencies on terms "7" and "24". However,*

Algorithm 5 $versionDecision()$

Require: queryAdjunctList

$status = -1$

p(y) = $\exists y$ s.t. (y is a version)$\wedge$(has all positive distances assigned to it)

p‘(y) = $\exists y$ s.t. (y is a version)$\wedge$(has all negative distances assigned to it)

$\forall query$ s.t. $query \in queryAdjunctList$
if (p(y) $\wedge \neg$ p‘(y) **then**
    $status = 1$
else if ((p‘(y) $\wedge \neg$ p(y)) **then**
    $status = 2$
else
    $status = 3$
end if
Define: $probScore = \frac{1}{distance * depth}$
for ($query1 \in queryAdjunctList$) **do**
    for ($query2 \in queryAdjunctList$) **do**
        if ($query1.possibleVersions \cap query2.possibleVersions \not\leftarrow \emptyset$) **then**
 # pVs = possibleVersions #
 # pV ∈ pVs #
            for all ($query1.pV = query2.pV$) **do**
                if ($status = 1$) **then**
                    if (($query1.pV.distance < 0$) $\vee$ (($query1.pV.distance > 0$) $\wedge$ ($query2.pV.distance > 0$) $\wedge$ ($query1.pV.probScore < query2.pV.probScore$))) **then**
                        $query1.pVs \leftarrow query1.pVs \setminus query1.pV$
                    end if
                else if ($status = 2$) **then**
                    if (($query1.pV.distance > 0$) $\vee$ (($query1.pV.distance < 0$) $\wedge$ ($query2.pV.distance < 0$) $\wedge$ ($query2.pV.probScore < query1.pV.probScore$))) **then**
                        $query1.pVs \leftarrow query1.pVs \setminus query1.pV$
                    end if
                else
                    if ($query1.pV.probScore < query2.pV.probScore$) **then**
                        $query1.pVs \leftarrow query1.pVs \setminus query1.pV$
                    end if
                end if
            end for
        end if
    end for
end for

due to their closeness to their software counterparts “7" gets assigned to “Windows" and “24" gets assigned to “Firefox".

Once the versions have been assigned to the software queries, it is required to remove redundant version assignments. For example, in Fig. 4.7 version “420" is assigned to all the queries except the first “Centrify". Since a version can only be assigned to a single software name, it is important to remove such redundant assignments. Heuristics 7 and 8 are aimed at removing redundant version assignments and are realized in Algorithm 5.

Algorithm 5 begins by assigning a *version alignment status* to the sentence. The three statuses (1,2 and 3) reflect the three alignment concepts presented in heuristic 7 in the same order. The status of the example vulnerability description used in this section is 1. This can be inferred by observing Fig. 4.8, where version “2008" (first version term) has a single occurrence with a positive distance and no other version term (including the last occurring version term) has an all negative list of distances. Algorithm 5 also defines a new variable *probScore*. The probScore is generated on the basis of heuristics 8 and is calculated as $\frac{1}{distance*depth}$. Distance and depth are terms introduced in the previous section (4.2.5). Essentially probScore is a measure of the closeness of a version term to the software terms it is assigned to.

Redundant version assignments are removed using *version alignment status*-es and rules created from heuristics 7 and 8. The rules state that while comparing two similar version assignments if the *version alignment status* is 1 and one of the version distances is negative, the association of that version is removed from its corresponding software. In Fig. 4.8 software query “DirectControl" looses its association with version term “20141" for the above mentioned reason. However, if both distances are positive then the software query having maximum *probScore* keeps the version. Again from Fig. 4.8 version term “3x" looses its association with software queries “Server Suite", “Centrify", “LINUX" and “UNIX" as query “DirectControl" manifests a higher probScore (1) than all of the above mentioned queries. If *version alignment status* is 2, the exact opposite operations are performed. Finally if *version alignment status* is 3, then the distinction is made solely on the basis of *probScore*. The version assignment having a lower *probScore* is removed.

Algorithm 6 $getModifiers()$

Require: queryAdjunctList, sentence-bundle

for ($queryAdjunct \in queryAdjunctList$) **do**
  for ($dependecy \in sentence\text{-}bundle.resultantDependencyList$) **do**
  # dependency = ⟨relation, governor, dependent⟩ #
    b = (governor $\in$ $queryAdjunct.possibleVersion$) $\wedge$ (dependent $\in$ $queryAdjunct.possibleVersion$)
    if (($relation =' prep\_*') \wedge b$) **then**
      $modifier1 \leftarrow 'governor + " " + relation + " " + dependent'$
      $modifier2 \leftarrow 'relation + " " + dependent'$
      if ($modifier2 \in sentence\text{-}bundle.resultantHalf$) **then**
        $queryAdjunct \leftarrow queryAdjunct \cup modifier2$
      else if ($modifier1 \in sentence\text{-}bundle.resultantHalf$) **then**
        $queryAdjunct \leftarrow queryAdjunct \cup modifier1$
      end if
    else if (($relation.relationName =' mark') \wedge b$) **then**
      $modifier \leftarrow 'dependent + " " + governor'$
      if ($modifier \in sentence\text{-}bundle.resultantHalf$) **then**
        $queryAdjunct \leftarrow queryAdjunct \cup modifier$
      end if
    else if (($relation.relationName =' conj') \wedge b$) **then**
      $modifier \leftarrow 'governor + " " + relation + " " + dependent'$
      if ($modifier \in sentence\text{-}bundle.resultantHalf$) **then**
        $queryAdjunct \leftarrow queryAdjunct \cup modifier$
      end if
    end if
  end for
end for

4.2.7 Identifying Modifiers

Heuristic 9. *A modifier is essentially a single typed dependency in which either both the governor and the dependent or just the dependent are version terms. The typed dependency is chosen from the following list:*

- *Prepositions (prep)*
- *Marker (mark)*
- *Conjunctions (conj)*

Software	Versions	Modifiers
Centrify Server Suite	2008,20141	
Centrify DirectControl	3x, 420	3x through 420
Linux		
UNIX		

Figure 4.9: Final Output

For the sake of readability modifiers are expressed as a concatenation of the following strings: governor (if a version term), relation, dependent.

In the process of extracting modifiers, all dependencies from the *resultantDependencyList* component of the sentence-bundle are scanned and if the condition expressed in heuristic 9 is satisfied, a modifier is formed by combining the governor, relation and dependent as shown in Algorithm 6. The combination is dependent on the way dependencies are expressed by the Stanford Typed Dependency representation. For example, prepositional dependencies are expressed as `conj_*(governor,dependent)`, where * can refer to conjunctions like “and" or “or". Since by heuristic 9 both the governor and dependent are versions, the modifier is represented as “governor * dependent" where * refers to the conjunction. Once the modifier string is formed, it is

verified whether the input sentence (*sentence-bundle.resultantHalf*) contains the string. If it does, the modifier is added to the query tuple (*queryAdjunct*).

From Fig. 4.4 versions “3x" and “420" (both assigned to query “DirectControl" in Fig. 4.8) are dependent on each other by the relation "prep_through". As a result, the modifier is represented as “3x through 420".

4.2.8 Post-Processing

Finally, software queries which belong to the same NNPgroup are merged into a single software name if they exist side-by-side in their group. Their versions and modifiers are also aggregated. As an example, queries “Centrify" and “Server Suite" from Fig. 4.7 are merged into one single software name “Centrify Server Suite" (as they exist side-by-side in an NNPgroup) as shown in Fig. 4.9. Similarly, queries “Centrify" and “DirectControl" are merged into a single software name “Centrify DirectControl".

4.3 Extracting Attacker Actions, User Actions and Post-Conditions (Component 2)

Component 2 takes a more logical modus operandi than component 1. The approach requires an in-depth analysis of independent English clauses, the workflow of which is shown in Fig. 4.10. Similar to the approach taken in component 1, we begin by creating a sentence-bundle. However, a sentence-bundle in this section, emphasizes more on the grammatical aspects of a vulnerability description. It contains a parts-of-speech tagged syntax tree of the vulnerability description and a list of linguistic dependencies within the constituent words of the description. Both these contents are required extensively in the subsequent phases of component 2. The basic theory behind the subsequent phases is founded upon the following two points:

1. *Attackers* and *users* are *humans*.

2. *Actions* and *impacts* (*post-conditions*) are effected by *humans*.

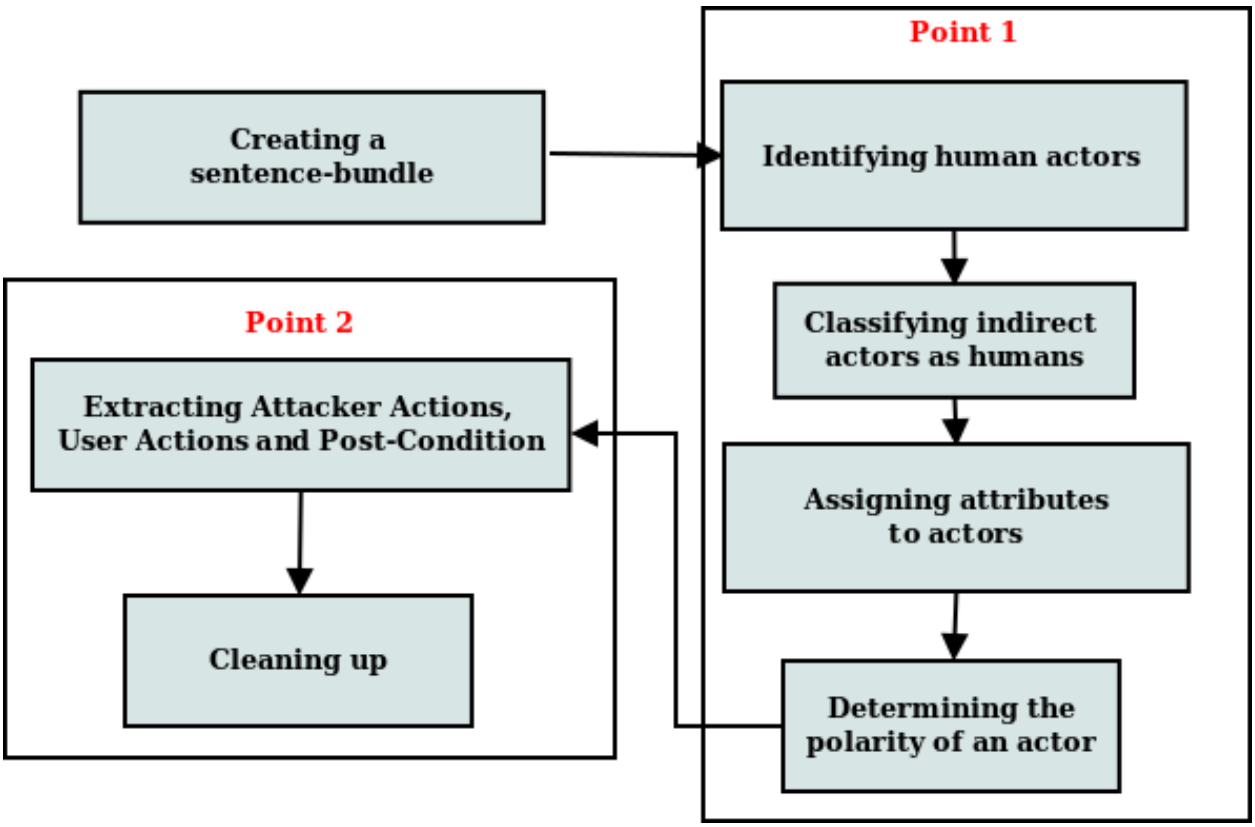


Figure 4.10: Component 2 Workflow Diagram

Point 1 from the above enumeration is realized using the 4 phases (subsections) of this approach and shown in Fig. 4.10. At first, human actors are identified in phase `Identifying human actors` (subsection 4.3.2). Due to the nature of English language, often actors (subjects) in a sentence are mentioned implicitly or indirectly. For example, in the sentence “The software was compromised", the subject (possibly an attacker) is implicit. We refer to such subjects as *indirect actors*. However, since these actors are not referenced directly in the text it is required to verify whether they refer to human subjects. This issue is dealt with in phase `Classifying indirect actors as humans` (subsection 4.3.3). Once human actors are identified, the next requirement is to check whether they are *attackers* (performing actions with negative intents) or *users* (performing actions with positive or neutral intents). Humans can often be classified as malicious or benign on the basis of character traits and actions performed. In other words, by determining the polarity of adjectives and verbs associated with the concerned actor (subject) we

can attempt to classify the actor as an attacker or a user. For this purpose, we first assign attributes (adjectives and verbs) to each identified human actor in phase `Assigning attributes to actors` (subsection 4.3.4) and then classify the actors as negative (attacker) and positive or neutral (user) by determining the polarity of the assigned attributes in phase `Determining the polarity of an actor` (subsection 4.3.5).

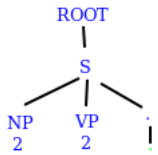
While the boxes (phases) shown in Fig. 4.10 under the point 1 red contour focus on finding words which refer to attackers and users in the input vulnerability descriptions, those shown under the point 2 red contour focus on extracting parts of the vulnerability descriptions which refer to actions performed by both attackers and users or goals (post-conditions) achieved by attackers. Point 2 is realized using the phases `Extracting` *`Attacker Actions`*`,` *`User Actions`* `and` *`Post-Conditions`* (subsection 4.3.6) and `Cleaning up` (subsection 4.3.7). While the purpose of the first phase can be gauged from its heading/name, the second phase attempts at removing redundant action and post-condition assignments to attackers and users.

To better explain each phase/subsection shown in Fig. 4.10, we would be using the sanitized (refer to section 4.1) version of the vulnerability description exampled below (**source :** NVD(CVE-2010-0483)).

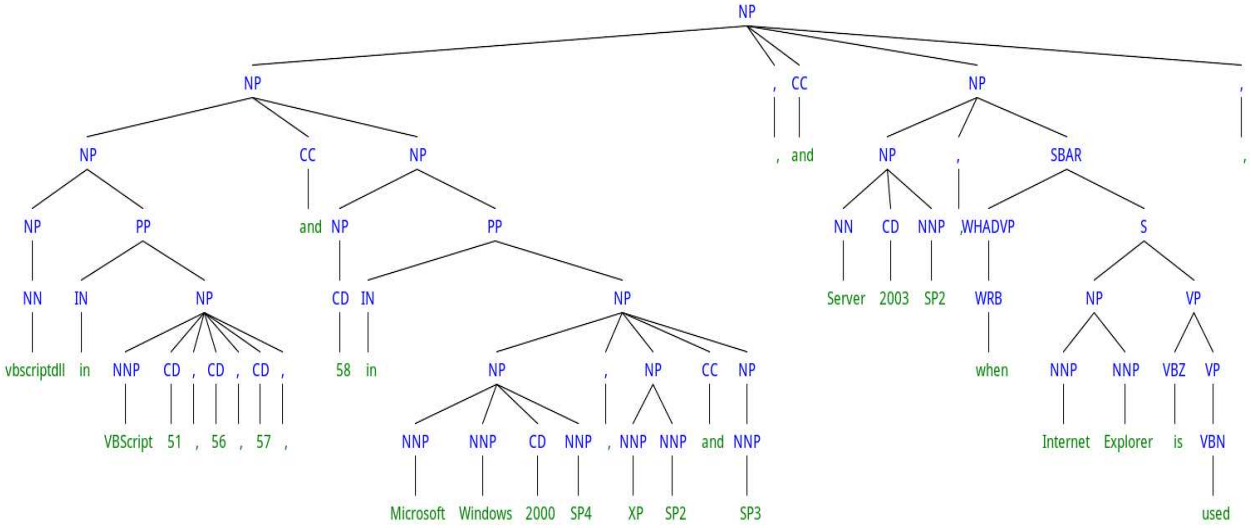
> *"vbscriptdll in VBScript 51, 56, 57, and 58 in Microsoft Windows 2000 SP4, XP SP2 and SP3, and Server 2003 SP2, when Internet Explorer is used, allows user-assisted remote attackers to execute arbitrary code by referencing a (1) local pathname, (2) UNC share pathname, or (3) WebDAV server with a crafted hlp file in the fourth argument (aka helpfile argument) to the MsgBox function, leading to code execution involving winhlp32exe when the F1 key is pressed. "*

4.3.1 Creating a Sentence-Bundle

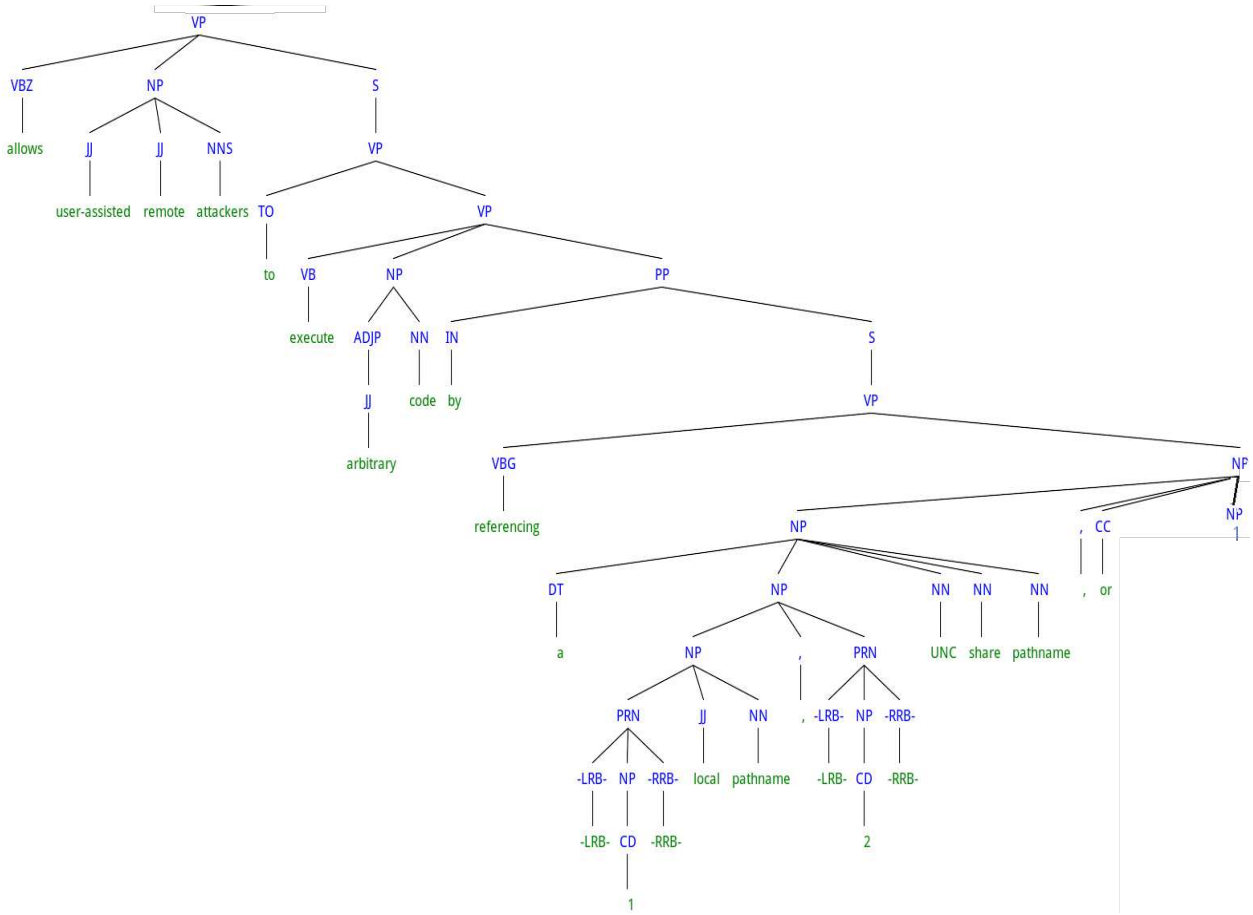
We begin component 2 by creating a sentence-bundle. The purpose of creating a sentence-bundle in component 2 is distinctly different from that of component 1. As a result, we redefine the concept of a sentence-bundle in this section.



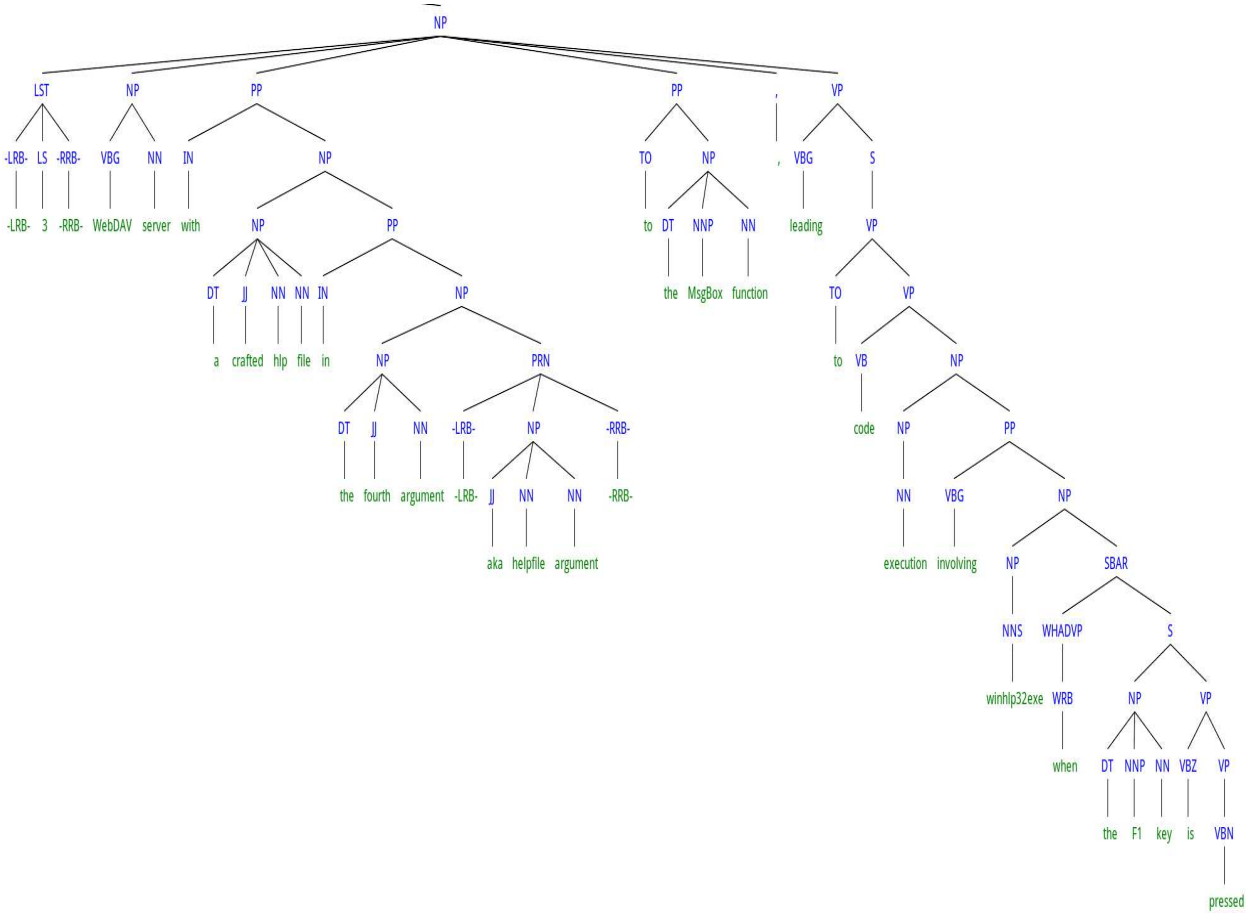
(a) Top Three Levels



(b) NP-2 Subtree



(c) VP-2 Subtree



(d) NP-1 Subtree

Figure 4.11: Syntax Tree for Vulnerability Description [CVE-2010-0483] (*synTree*)

Definition (Sentence-Bundle)**.** *A sentence-bundle is a formatted grammatical representation of a sentence and is used for the purpose of extracting Attacker Actions, User Actions and Post-Conditions. A sentence-bundle can be viewed as a 2-tuple, the elements of which are a parts-of-speech tagged syntax tree and the list of grammatical dependencies in an input sentence.*

Fig. 4.11 and Table 4.1 represent the constituent syntax tree (synTree) and dependency list (dList) of the sentence-bundle created for the example vulnerability description used in this section. The syntax tree is shown in Fig. 4.11. Due to its substantial size, it is split into four subtrees, each outlined in 4 different sub-figures. Fig. 4.11a represents the top three levels of the syntax tree. The root node is labeled using the default “ROOT" tag. The successor of the root node, the “S"-labelled node, represents the primary declarative/independent clause of the sentence. By its general

Table 4.1: Grammatical Dependency List for Vulnerability Description [CVE-2010-0483] (dList)

relation	governor	governor-pos	dependent	dependent-pos	relation	governor	governor-pos	dependent	dependent-pos
nsubj	allows	34	vbscriptdll	1	prepc_by	execute	39	referencing	43
xsubj	execute	39	vbscriptdll	1	det	pathname	56	a	44
prep_in	vbscriptdll	1	VBScript	3	appos	pathname	49	1	46
num	VBScript	3	51	4	amod	pathname	49	local	48
num	VBScript	3	56	6	nn	pathname	56	pathname	49
num	VBScript	3	57	8	appos	pathname	49	2	52
conj_and	vbscriptdll	1	58	11	nn	pathname	56	UNC	54
nsubj	allows	34	58	11	nn	pathname	56	share	55
xsubj	execute	39	58	11	dobj	referencing	43	pathname	56
nn	SP4	16	Microsoft	13	dep	server	63	3	60
nn	SP4	16	Windows	14	amod	server	63	WebDAV	62
num	SP4	16	2000	15	dobj	referencing	43	server	63
prep_in	58	11	SP4	16	conj_or	pathname	56	server	63
nn	SP2	19	XP	18	det	file	68	a	65
prep_in	58	11	SP2	19	amod	file	68	crafted	66
conj_and	SP4	16	SP2	19	nn	file	68	hlp	67
prep_in	58	11	SP3	21	prep_with	server	63	file	68
conj_and	SP4	16	SP3	21	det	argument	72	the	70
nn	SP2	26	Server	24	amod	argument	72	fourth	71
num	SP2	26	2003	25	prep_in	file	68	argument	72
conj_and	vbscriptdll	1	SP2	26	amod	argument	76	aka	74
tmod	used	32	SP2	26	nn	argument	76	helpfile	75
nsubj	allows	34	SP2	26	appos	argument	72	argument	76
xsubj	execute	39	SP2	26	det	function	81	the	79
advmod	used	32	when	28	nn	function	81	MsgBox	80
nn	Explorer	30	Internet	29	prep_to	server	63	function	81
nsubjpass	used	32	Explorer	30	vmod	server	63	leading	83
auxpass	used	32	is	31	aux	code	85	to	84
rcmod	SP2	26	used	32	xcomp	leading	83	code	85
root	ROOT	0	allows	34	dobj	code	85	execution	86
amod	attackers	37	user-assisted	35	prep_involving	execution	86	winhlp32exe	88
amod	attackers	37	remote	36	tmod	pressed	94	winhlp32exe	88
dobj	allows	34	attackers	37	advmod	pressed	94	when	89
aux	execute	39	to	38	det	key	92	the	90
xcomp	allows	34	execute	39	nn	key	92	F1	91
amod	code	41	arbitrary	40	nsubjpass	pressed	94	key	92
dobj	execute	39	code	41	auxpass	pressed	94	is	93
rcmod	winhlp32exe	88	pressed	94					

definition, a declarative clause consists of a subject part (noun phrase) and a predicate part (verb phrase). Sub-trees representing the subject and predicate part of the primary declarative clause are shown in Fig. 4.11b and Fig. 4.11c respectively. However, due to the lack of printing space, the verb phrase sub-tree (Fig. 4.11c) is continued in Fig. 4.11d. It is to be noted that some sub-figures in Fig. 4.11 contain nodes marked with numbers (NP1, VP2 etc). These nodes are later expanded into sub-trees in subsequent sub-figures. Table 4.1 lists the grammatical dependencies found in the example vulnerability description used in this section. Each dependency is represented using a set of 5 columns namely, *relation*, *governor*, *governor-pos*, *dependent* and *dependent-pos*. The term “pos" refers to the position of a governor or dependent after the sentence is tokenized

4.3.2 Identifying Human Actors

We begin the extraction process by defining the concept of a human actor.

Definition (Human Actor)**.** *Human actors are entities that perform certain actions and are depicted as common noun objects in vulnerability descriptions. Human actors can include entities that do not refer to humanoid objects. For example, terms like "server" and "machine" can be treated as human actors. Human actors can be subdivided into the following categories:*

- ***Attackers****: Those entities that perform security-critical actions with negative or malicious intents.*
- ***Users****: Those entities that perform security-critical actions with positive or benign intents. They are often the victim of targeted attacks.*
- ***Others****: Those entities that perform actions with positive or benign intents. The actions performed by these entities are not security critical and hence are not related to the PAG concepts.*

Heuristic 10. *In a sentence, human actors are expressed using words that appear as dependent-s for the following set of relations:*

- ***Nominal Subject*** *(nsubj)*
- ***Agent*** *(agent)*
- ***Direct Object*** *(dobj)*
- ***Passive Nominal Subject*** *(nsubjpass)*

Heuristic 11. *In a sentence, human actors are expressed using words which contain the following singular terms in their dictionary meanings:*

- *Third Person Relative Pronouns*

Algorithm 7 $getHumanActors()$

Require: sentence-bundle
Ensure: governor of the dependency is a verb and dependent is a noun
$x \leftarrow$ {nsubj, agent, dobj}
$y \leftarrow$ {nsubjpass}
$keywords \leftarrow$ {“who", “whose", “anyone", “anybody", “someone", “somebody"}
1: **for** $(dependency \in sentence\text{-}bundle.dList)$ **do**
 # dependency = ⟨ relation, governor, dependent ⟩ #
2: **if** $(relation \in x)$ **then** *# Direct Actor #*
3: **if** (tokenized wordNetGloss ∩ keywords) **then** *# wordNetGloss = dictionary meaning of a word #*
4: $directActorList \leftarrow directActorList \cup dependent$
5: **else**
6: $nonHumanActorList \leftarrow nonHumanActorList \cup dependent$
7: **end if**
8: **else if** $(relation \in y)$ **then** *# Indirect Actor #*
9: $indirectActorList \leftarrow indirectActorList \cup dependent$
10: **end if**
11: **end for**
12: **return** *directActorList*, *indirectActorList*, *nonHumanActorList*

  - *“who"*
  - *“whose"*

- *Third Person Indefinite Pronouns*
  - *“anybody"*
  - *“anyone"*
  - *“somebody"*
  - *“someone"*

In Algorithm 7 we traverse the grammatical dependencies listed in Table 4.1 and attempt to find dependencies which incorporate the relations listed in heuristic 11. For dependencies which relate to nominal subjects, agents or direct objects we consider the dependent nouns as `direct actors` if the tokenized version of their dictionary meanings contain words listed in heuristic 12. On the other hand, if dependencies relate to passive nominal subjects we consider their depen-

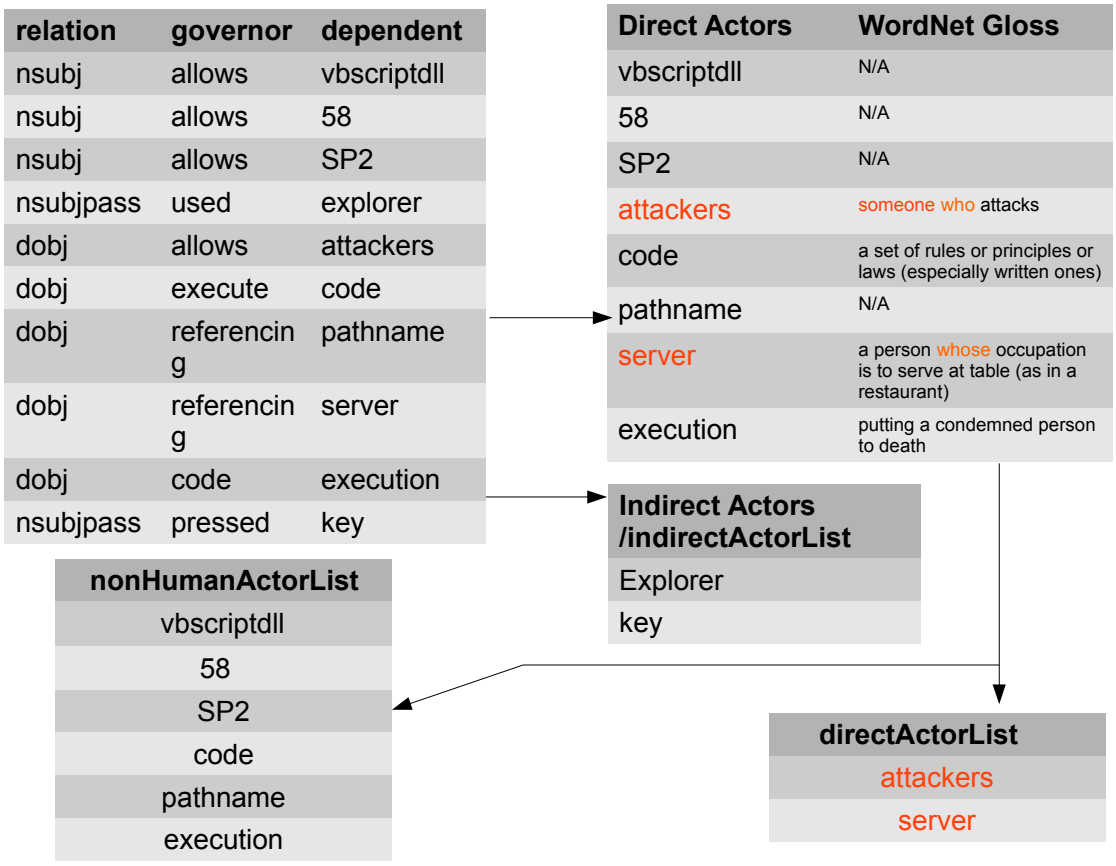


Figure 4.12: Identifying Human Actors

dents as `indirect actors`, regardless of whether their dictionary meanings refer to humans. Dictionary meanings are retrieved as wordnet glosses (see section 3.3 for further details).

Fig. 4.12 shows the workflow as explained in Algorithm 7. Once dependencies with matching relations (tabulated on the left) are found, either the dependent's wordnet glosses are used to confirm their status as humans (direct actors) or simply added as indirect actors. A notable inclusion in the *directActorList* from Fig. 4.12 is the word "server". Although in computer science vocabulary "server" generally refers to a machine, in orthodox English it refers to "someone who serves" which justifies its inclusion in the list of human actors. Apparently, such an inclusion satisfies our needs since "server" machines can also perform activities which lead to security compromises.

Algorithm 8 $areIndirectActorsHuman()$

Require: sentence-bundle,indirectActorList, directActorList, nonHumanActorList

for all ($indirectActor \in indirectActorList$) **do**
 # indirectActorDependency = the original dependency from where the indirect actor was obtained #
    $isNonHumanActor \leftarrow false$
    $enqueue\ indirectActorDependency$ *# Add to a queue #*
    while ($queue\ is\ not\ empty$) **do**
        $currentDependency = dequeue$ *# Remove an element from the bottom of the queue #*
        for all ($dependency \in sentence\text{-}bundle.dList$) **do**
 # dependency = ⟨governor,dependent,relation⟩ #
            if (verb component of $dependency$ = verb component of $currentDependency$) **then** *# verb component refers to either the governor or the dependent or both if they are verbs #*
                if ($dependent \in nonHumanActorList$) **then** *# nonHumanActorList is obtained from Algorithm 7 #*
                    $indirectActorList = indirectActorList \setminus indirectActor$
                    EndLoop
                else
                    $enqueue\ dependency$
                end if
            end if
        end for
    end while
end for
$directActorList \leftarrow directActorList \cup indirectActorList$

4.3.3 Classifying Indirect Actors as Humans

Heuristic 12. *Indirect actors are non-human if they are directly or indirectly grammatically related to other non-human actors.*

The inclusion of indirect actors engenders an ambiguity about their nature, human or not. Indirect actors are not included in the input vulnerability description and are therefore indiscernible in context of the sentence. Consequently, it is hard to determine whether they refer to human nouns. Heuristic 13 is designed to resolve this issue. Algorithm 8 is used to formally realize the concept established in heuristic 13. It is to be noted that in absence of an actual mention of the indirect actor in the vulnerability description, we consider dependents of `nsubjpass` relations as provisional indirect actors.

Table 4.2: Types of Attributes Assigned to Each Actor

Indirect Actors	*Verbs*	governing verb in nsubjpass, open clausal complements of governing verb (`xcomp`)
	Modifiers	adjective modifiers for direct object of governing verb (`amod`)
Direct Actors	*Verbs*	governing verb in `nsubj` or `agent`, reduced non-finite verbal modifiers of the dependent (`vmod`), open clausal complements of the governing verb (`xcomp`)
	Modifiers	adjective modifiers for direct objects of the governing verb (or of `xcomp` of the governing verb, if actor was an object) (`amod`), adjective modifiers of the dependent (`amod`)

In Algorithm 8 we traverse the list of indirect actors and for each indirect actor, we add *indirectActorDependency*-s (dependencies from which indirect actors were obtained) to a queue. At every iteration we dequeue elements from the queue and perform the following tasks:

1. If the verb component (check Algorithm 8 for reference) of the dequeued dependency matches to the verb component of any dependency from Table 4.1, we buffer the dequeued dependency.

2. The dependent component of the buffered dependency is then checked against the *nonHumanActorList*. If a match is found, we assume the indirect actor to be non-human and remove it from the list of indirect actors (*indirectActorList*). Otherwise, we add the buffered dependency to the queue and continue until the queue is empty.

Eventually, the list of indirect actors (holding only those actors which are assumed to be human) is appended to the list of human actors.

For the example used in this section, indirect actors “key" and “Explorer" (refer to Fig. 4.12) do not have any direct or indirect grammatical links to non-human actors (also shown in Fig. 4.12). As a result, the indirect actors, not mentioned in the text, are considered to be human nouns.

4.3.4 Assigning Attributes to Actors

Heuristic 13. *Human actors can be classified as attackers, users and others based on their actions (verbs) and features (adjective modifiers or simply modifiers). The types of verbs and modifiers used in the characterization process are tabulated in Table 4.2.*

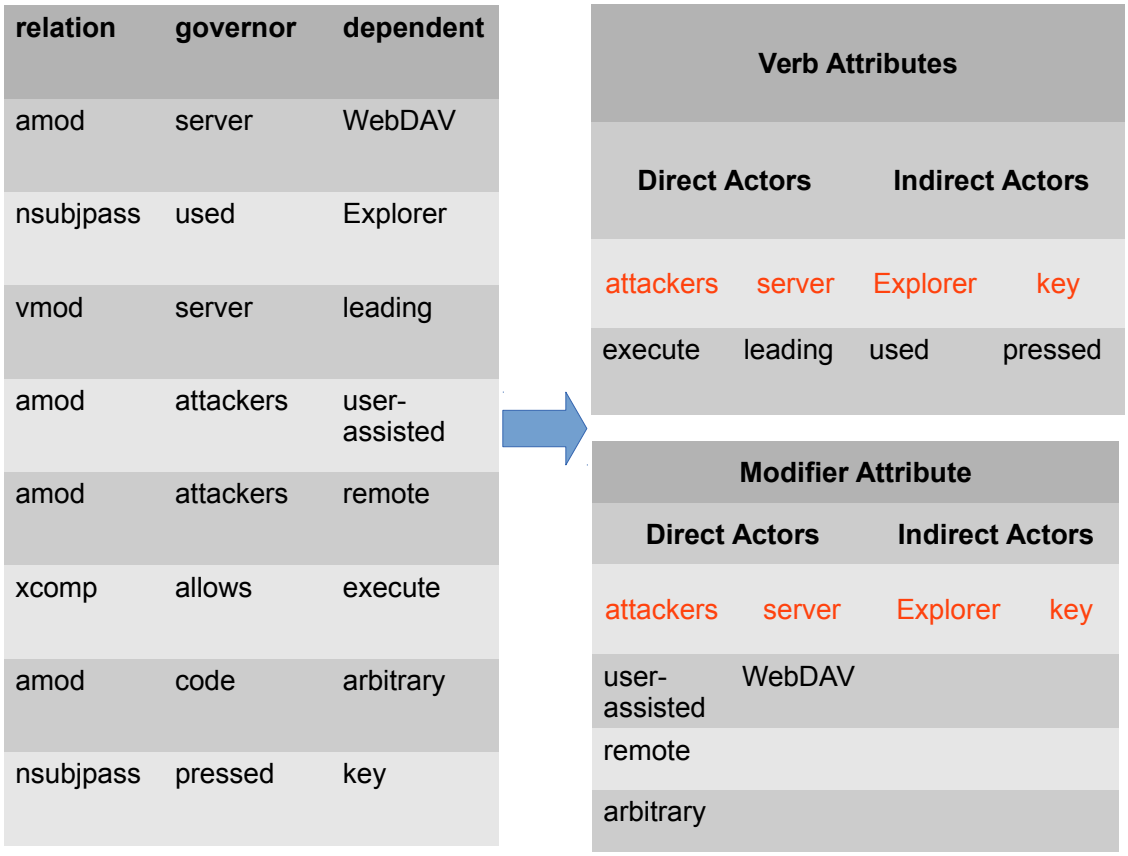
After the list of human actors is identified, each actor is associated with a set of attributes. These attributes are directly or indirectly related to the actor. Attributes are assigned in accordance with heuristic 13. For the sake of simplicity, we assume the attributes to be stored in the data-structure *attributeList*.

Fig. 4.13 is an example of how attributes are assigned to each actor. The dependencies from which these attributes are extracted are shown on the left-hand side of Fig. 4.13. Some attribute linkages are presented directly. For example, the linkages between “server" and “leading" (vmod), “key" and “pressed" (governing verb of nsubjpass) and “attackers" and“user-assisted" (amod) are direct. However, some linkages are indirect and cannot be directly observed in Fig. 4.13. For example, the linkage between “attackers" and “execute" shown under the `Verb Attributes` table cannot be observed in the list of dependencies shown on the left. However “execute" is the “xcomp" modifier of “allows" which objectifies the term “attacker". Hence its inclusion.

4.3.5 Determining the Polarity of an Actor

Heuristic 14. *Human actors can be classified as attackers, users and others using the following observations:*

- *Human actors referred to as “attackers" in the input vulnerability description or having at least one negatively polarized attribute are attackers.*
- *Human actors are users if they have they exhibit the following features*
  - *They are referred to as “victim[s]" in the input vulnerability description.*
  - *They have no negatively polarized attribute and are referred to as “user[s]" in the input vulnerability description.*



relation	governor	dependent
amod	server	WebDAV
nsubjpass	used	Explorer
vmod	server	leading
amod	attackers	user-assisted
amod	attackers	remote
xcomp	allows	execute
amod	code	arbitrary
nsubjpass	pressed	key

Verb Attributes			
Direct Actors		**Indirect Actors**	
attackers	server	Explorer	key
execute	leading	used	pressed

Modifier Attribute			
Direct Actors		**Indirect Actors**	
attackers	server	Explorer	key
user-assisted	WebDAV		
remote			
arbitrary			

Figure 4.13: Attributes Assigned to each Actor

  - *They have no negatively polarized attribute and are indirect actors.*

- *Human actors with no negatively polarized attribute are others if they are direct actors and are referred by anything other than“victim[s]" and “user[s]" in the input vulnerability description.*

The word "attacker" is often used to refer to a malicious individual whose goal is to compromise a system. But this can be deceptive. On certain occasions, malicious adversaries can be "user"s or even "server"s. As an instance, in the following vulnerability description [**source :** NVD (*CVE-2010-0237*)] the word “user" is used to depict the malicious adversary.

> *“The kernel in Microsoft Windows 2000 SP4 and XP SP2 and SP3 allows local **users** to gain privileges by creating a symbolic link from an untrusted registry hive to a trusted registry hive, aka "Windows Kernel Symbolic Link Creation Vulnerability.""*

It is thus not correct to assign a positive polarity to all instance of the word “user". In other words, dictionary meanings are not enough to classify actors as malicious or benign. We need to assess the context in which they appear in the vulnerability descriptions. In this thesis, the context is represented in terms of the attributes assigned to each actor in the previous (4.3.4) section. According to heuristic 14 determining the polarity of the attributes, and text-form representations of human actors can aid us in classifying actors as attackers or users. Algorithm 9 is used formally realize heuristic 14.

In Algorithm 9 we first check the textual representation of the entity in the input description. If the entity is referred to as “attacker" or "victim" in the original description, our algorithm ends by assigning this entity as an attacker or user respectively. Otherwise, we obtain polarity values from SentiWordNet 3.0 (refer to section 3.4) and if any of attributes return a negative polarity, we refer to the actor as "attacker". Conversely, if the actor is indirect we refer to it as “user" or “other" if it is direct.

Fig. 4.14 shows an example implementation of Algorithm 9. Out of the four actors identified in the previous stages, the actor “attacker" is easily classified as an attacker due to its textual depiction. The actors “server", “keyword" are indirect and have all positively polarized attributes (“used" and “pressed"). Hence the indirect actors they represent are classified into the user category. Finally, the actor “server" is categorized into other human actors as it is neither indirect nor has a negatively polarized attribute assigned to it.

4.3.6 Sectioning Into Attacker Actions, User Actions and Post-Conditions

In this step we extract instances of *Attacker Action*s, *User Action*s and *Post-Condition*s from the input vulnerability descriptions. To completely comprehend the extraction process the reader needs to be familiarized first with the concept of a `minimal phrase` or a `minimal clause`.

Definition (Minimal Phrase/Clause)**.** *The minimal phrase or clause, for a given word and a given tag, is a phrase/clause which cannot be subdivided into a phrase/clause of similar tag containing the same word.*

Algorithm 9 $determineActorType()$

Require: directActorList, indirectActorList

for all ($directActor \in directActorList$) **do**
  if ($directActor = caseInsensitve(\text{“}attacker[s]\text{”})$) **then**
    directActor is $attacker$
  else if ($directActor = caseInsensitve(\text{“}victim[s]\text{”})$) **then**
    directActor is $user$
  else
    $isAttacker \leftarrow false$
    for ($attribute \in directActor.attributeList$) **do**
      $sentimentScore \leftarrow getSentimentScorefromSentiWordNet(attribute)$
      if ($sentimentScore < 0$) **then**
        directActor is $attacker$
        $isAttacker \leftarrow true$
        end loop
      end if
    end for
    if ($isAttacker\ is\ false$) **then**
      if $((directActor \in indirectActorList) \wedge (directActor = caseInsensitve(\text{“}user[s]\text{”}))$ **then**
        directActor is $user$
      else
        directActor is $other$
      end if
    end if
  end if
end for

• ***Example*** *: The parse tree of the sentence “The attacker broke into the machine to steal data." is shown below.*

```
(ROOT
(S((NP(DT The)(NN attacker))
(VP
 (VBD broke)(PP(IN into)(NP(DT the)(NN machine)))
 (S(VP(TO to)(VP(VB steal)(NP(NNS data))))))
)
(..)))
```

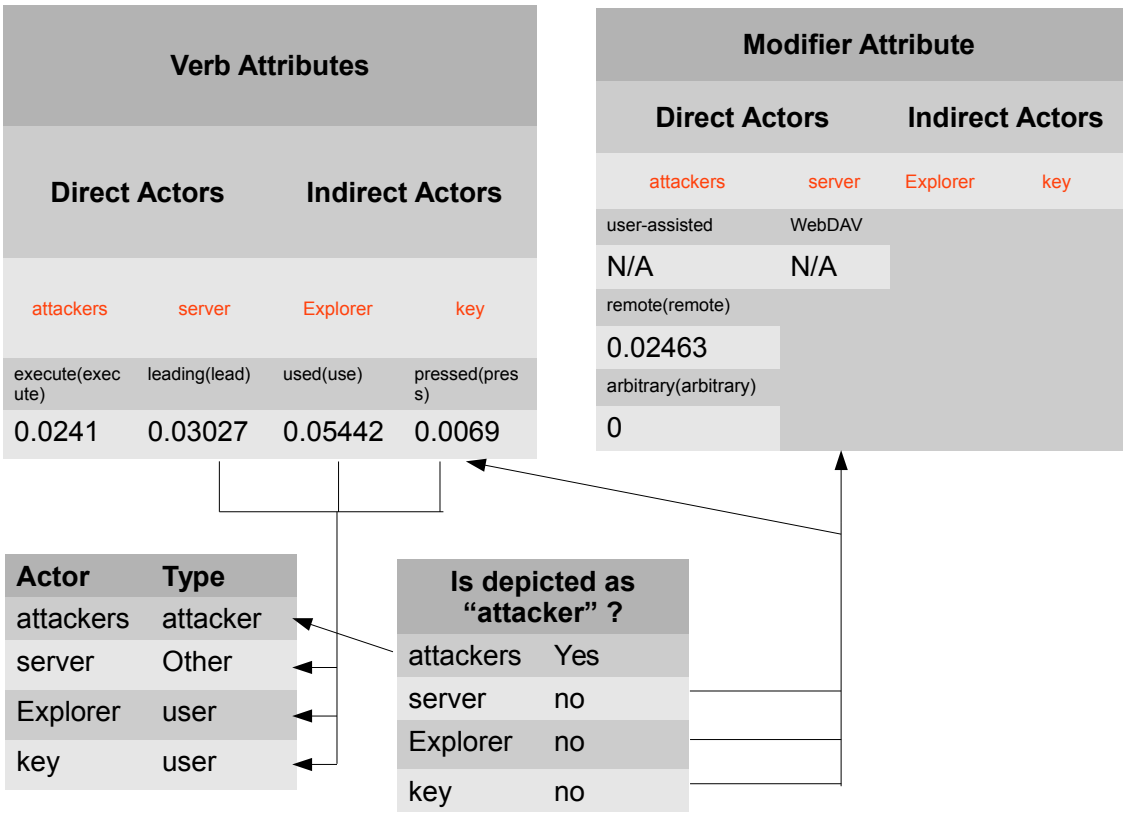


Figure 4.14: Determining the Type of Human Actor

The minimal verb phrase containing the word "broke" is the following phrase:

```
(VP(VBD broke)(PP(IN into)(NP(DT the)(NN machine)))
 (S(VP(TO to)(VP(VB steal)(NP(NNS data))))))
```

because if it is sub-divided into further verb phrases, they will not enclose the word "broke".

Heuristic 15. *Textual representations of Attacker Action have the following characteristics:*

- *They are verb phrases which are nested within minimal declarative clauses or verb phrases and begin with the gerund keywords: "using".*

- *They are preposition phrases which are nested within minimal declarative clauses or verb phrases and begin with the keywords: “by", “through", “with" and “via".*

Textual representations of User Action have the following characteristics:

- *They are minimal declarative clauses which enclose action verbs (governing verbs of nsubj, agent and nsubjpass relations).*
- *They are minimal verb phrases which enclose all verbs other than action verbs.*
- *They are verb phrases which are nested within minimal declarative clauses or verb phrases and begin with the gerund keywords: “using", “resulting", “causing" and “leading".*
- *They are preposition phrases which are nested within minimal declarative clauses or verb phrases and begin with the keywords: “by", “through", “with" and “via".*

Textual representations of Post-Conditions have the following characteristics:

- *They are minimal declarative clauses which enclose action verbs (governing verbs of nsubj, agent and nsubjpass relations).*
- *They are minimal verb phrases which enclose all verbs other than action verbs.*
- *They are verb phrases which are nested within minimal declarative clauses or verb phrases and begin with the gerund keywords: “resulting", “causing" and “leading".*

All minimal phrases and clauses used in the above characteristics enclose verb attributes assigned to the respective actors.

Algorithm 10 is used to execute heuristic 15. Firstly, for each actor, minimal declarative clauses and verb phrases are identified. Following this, all verb and preposition phrases nested within the previously extracted minimal clauses and phrases are extracted. The algorithm now divides into two sections. The first section deals with the nested verb phrases, which are then tested for the starting words “using" and “resulting". If the concerned actor is an “attacker", then for the first

Algorithm 10 $determineActionsPostConditionsforEachActor()$

Require: directActorList

for all ($directActor \in directActorList$) **do**
  for all ($verb \in directActor.VerbList$) **do**
    if ($verb \in governing\ verb\ for\ \{nsubj, agent, nsubjpass\}$) **then**
      $VP \leftarrow minimal\ declarative\ clause\ to\ which\ this\ verb\ belongs$
    else
      $VP \leftarrow minimal\ verb\ phrase\ to\ which\ this\ verb\ belongs$
    end if
    $nestedVPList \leftarrow \{nested\ verb\ phrases\ within\ VP\}$
    $nestedPPList \leftarrow \{nested\ preposition\ phrases\ within\ VP\}$
    for all ($VPn \in nestedVPList$) **do**
      if ($VPn\ startsWith$ "$using$") **then**
        if ($directActor\ isAttacker$) **then**
          $VPn.toString\ is\ attacker\ action$
        else if ($directActor\ isUser$) **then**
          $VPn.toString\ is\ action$
        end if
      else if ($VPn\ startsWith$ "$resulting$" or "$causing$" or "$leading$") **then**
        if ($directActor\ isAttacker$) **then**
          $VPn.toString\ is\ post\ condition$
        else if ($directActor\ isUser$) **then**
          $VPn.toString\ is\ action$
        end if
      end if
    end for
    for all ($PPn \in nestedPPList$) **do**
      if ($PPn\ startsWith$ "via" or "by' or "$through$" or "$with$") **then**
        $PPn.toString \leftarrow beginning\ of\ PPn.toString\ to\ end\ of\ VP.toString$
        $PPn.toString\ is\ attacker\ action$
      end if
    end for
    if ($directActor\ isAttacker$) **then**
      $VP.toString\ is\ post\ condition$
    else if ($directActor\ isUser$) **then**
      $VP.toString\ is\ action$
    end if
  end for
end for

word “using", the nested phrase is considered to be an *Attacker Action*, or else it is considered to be a *Post-Condition*. For both starting verbs if the actor is a “user", then the nested verb phrase is

considered to be a *User Action*. Similarly, if the nested preposition phrases start with “via", “by", “with" and “through" they are considered to be *Attacker Actions*. However, before being added to the list of attackers actions, each preposition phrase is extended until the end of the verb phrase they belong in. This decision is purely empirical and has no grammatical motivation credited to it. It allows us to include certain additional information which could be left out due to inaccurate generation of the syntax tree. Finally, the outer phrase is added to the *Attacker Actions*, if the actor concerned is an attacker or added to *User Actions*, otherwise.

In Fig. 4.15 we first identify the minimal verb phrases and declarative clauses for action verbs each direct/human actor is attributed to. Since actors “key" and “Explorer" were retrieved from an *nsubjpass* relation they are attributed to minimal declarative clauses. On the other hand, the actor “attacker" was retrieved as the dependent of a *dobj* relation. As a result, a minimal verb phrase is attributed to this actor. The nested preposition phrases are added as *Attacker Actions*. The minimal declarative clauses, however, do not nest other phrases. As a result, they are added as *User Actions*. Finally, the minimal verb phrase is added as the sole instance of *Post-Condition*.

4.3.7 Cleaning up

Heuristic 16. *Instances of one concept should be devoid of instances of another concept. This heuristic excludes the instances of Attacker Actions including other instances of Attacker Actions performed by the same actor. In this exceptional case, the smaller instances of Attacker Actions are considered irrelevant.*

- *Eg. Refer to Fig. 4.16.*

The clean-up procedure essentially removes concept instances which are sub-strings of larger concept instances. In other words, for all actors, and for all concepts we verify whether any of the instances are complete sub-strings of another. If so, we completely remove the smaller string from the larger string. Going by heuristic 16, we do not remove an attacker action from a different attacker action if both of them are performed by the same actor. Instead, we remove the smaller *attacker action* string from the final results. A different set of removed sections include the noun

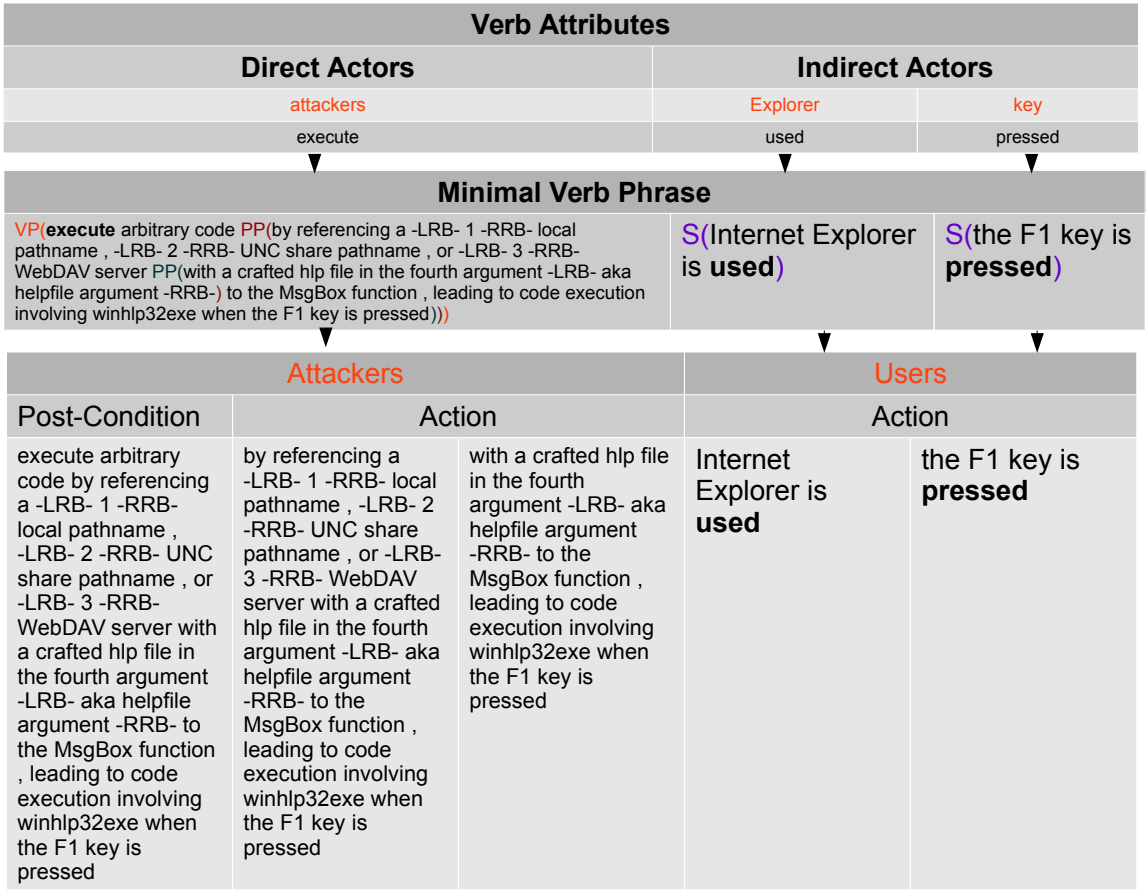


Figure 4.15: Segregating Actions and Post-Conditions

phrase that encloses words which refer to human actors. This is because, often times verb phrases might enclose a noun phrase enclosing the word which refers to the actor itself. The presence of this phrase does not provide any additional information apart and hence is removed. Finally, if extracted textual concepts start or end with common stop-words like “that", "and", "or" etc., white-space or punctuations, we remove them from the final result.

An example, of the clean-up procedure is shown in fig. 4.16. Textual concepts which are sub-strings of another concept are shown using arrows. The arrows are also labelled stating reason for clean-up. Since the same actor “attackers" is attributed to two actions and one is a substring of another, the smaller string is removed, leaving the larger string as the only *attacker action*.

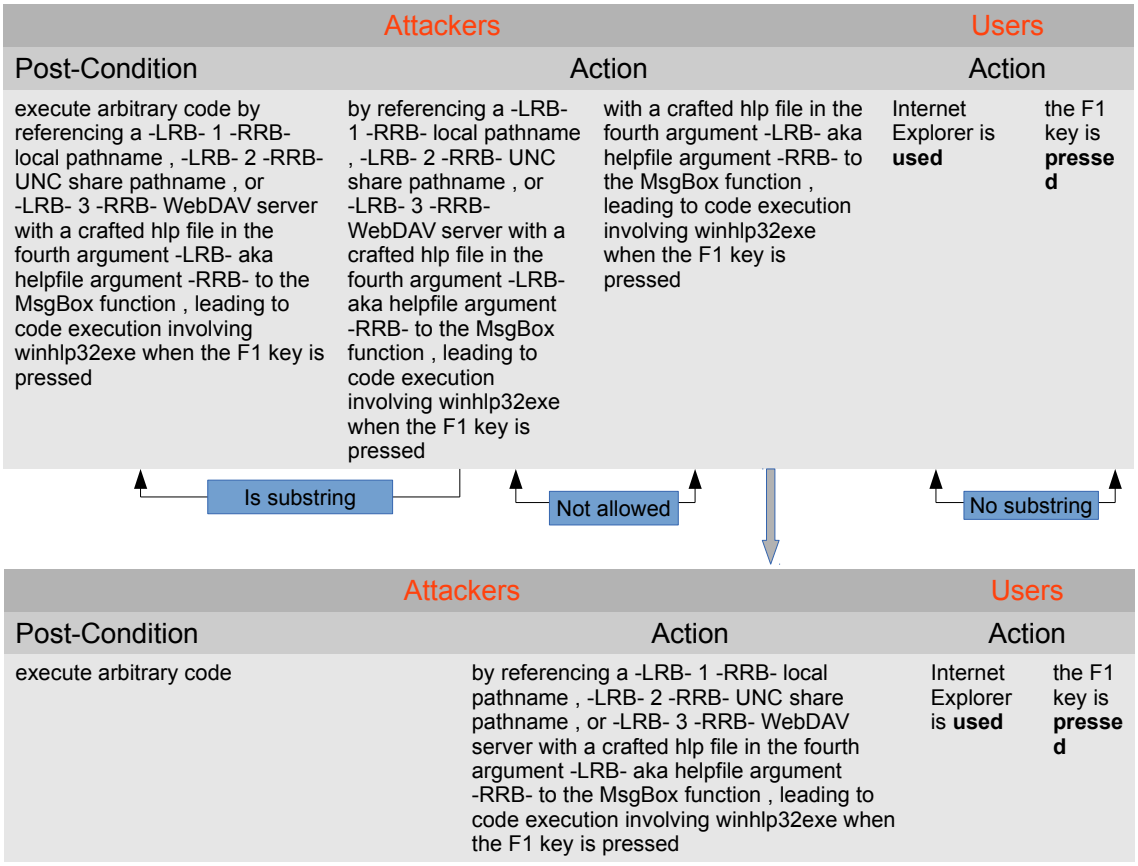


Figure 4.16: Clean up

Chapter 5

Evaluation and Discussion

The Personalized Attack Graph can be constructed by logically linking instances of the constituent concepts, otherwise referred to as the PAG concepts. Chapter 4 of this thesis describes an approach to automatically extract textual instances of the PAG concepts from vulnerability descriptions. The extraction process consists of two phases: extracting *Software Names*, *Versions* and *Modifiers* and extracting *Attacker Actions*, *User Actions* and *Post-Conditions*. Both these phases make use of the semantic consistencies observed in vulnerability descriptions written in natural language. However, owing to the fact that vulnerability descriptions are most often written and updated by humans, they are most often unstructured in nature and lack any form of consistency. Consequently, instances of PAG concepts extracted by the automated process might not always be correct and hence to evaluate the accuracy of the extraction process we need to compare automatically extracted concepts with their manually identified counterparts. In this chapter, we measure the performance of our approach by comparing the extracted concepts with those identified by human beings (having adequate knowledge of cyber security). We also compare the observed accuracy level with that obtained by Joshi et al. [13]. This allows us to perform a fair evaluation of our extraction techniques.

5.1 Evaluation Preliminaries

5.1.1 DataSets

Owing to the difference in judgement and bias introduced by divergent amount of knowledge on the topic, instances of the same concept identified by two different human subjects might not match. Consequently, the evaluation procedure needs to be performed on a corpus which satisfies the following criteria: it is not biased i.e. it is not generated by the author of this thesis and it is consistent i.e. generated by the collaborative effort of multiple subjects having adequate knowl-

Table 5.1: Joshi Corpus Classes vs PAG Concepts

PAG Concepts	Joshi Corpus Classes	Joshi Corpus Class Instances
Software Names and Versions	Software	Microsoft .Net Framework 3.5
	Operating Systems	Linux Ubuntu 10.4
	Hardware	IBM Mainframe B152
	Network_Terms	HTTP
Attacker Actions	Means	a crafted spreadsheet
Post-Conditions	Consequences	bypass intended access restrictions
User Actions		
	File_Name	index.php
	Other_Technical_Terms	HTML
Modifiers	NER_Modifier	through 7.5

edge of cyber-security. Two such labelled corpora of security related descriptions are available publicly [4, 13]. Of these, the `Bridges Corpus` [4] supports labels/classes which have very little resemblance with the PAG concepts. As a result we do not use this corpus to evaluate our approach. Instead we opt to use the `Joshi Corpus` [13].

Joshi Corpus

Table 5.1 compares the labelled classes/concepts in the Joshi corpus [11] to those in our corpus. From Table 5.1 it can be observed that:

1. Semantically similar classes are placed in adjacent columns. As an example, the class “Means" exudes an almost similar sense as the concept “Attacker Action". The same logic can be applied to the “Software Names and Versions" concept and “Software", “Operating System" classes. For simplicity we treat the “Hardware" class and the “Software Names and Versions" concept equivalently. The “NER_Modifier" class in the Joshi corpus provides version related information for software products.

2. It can be noticed that the Joshi Corpus does not include a class similar to the “User Actions" PAG concept. Similarly, the PAG does not make use of a “Network Terms" concept.

[11] `http://ebiquity.umbc.edu/resource/html/id/355`

Table 5.2: Joshi Corpus Concept Instance Distribution

	NVD	Adobe Security Bulletins	MS Security Bulletins	Security Blogs
Software Names and Versions	97	145	90	44
Modifiers	94	0	0	2
Attacker Actions	49	1	9	10
Post-Conditions	45	11	18	3

Table 5.3: Our Corpus Concept Instance Distribution

	Student1-Student2			Student1-Student3		
	NVD	**ISS-Xforce**	**SecurityFocus**	**NVD**	**ISS-Xforce**	**SecurityFocus**
Software	7	5	7	8	5	7
Version	6	0	2	5	0	2
Modifier	3	0	1	2	0	1
User Action	0	4	0	0	2	0
Attacker Action	4	4	2	4	3	0
Post-Condition	6	9	18	5	11	14

In-spite of subtle differences between the Joshi Corpus classes and the PAG concepts, the Joshi Corpus is the only publicly available corpus which is closest to our requirements. The archived version of the Joshi Corpus [12] contains a total of 395 vulnerability descriptions accumulated from the National Vulnerability Database (275), Adobe Security Bulletins (50), Microsoft Security Bulletins (50) and various security blogs (20). However, in their work [13], Joshi et al. use the following distribution to describe their corpus: 240 descriptions from the National Vulnerability Database, 30 security blogs and 80 security bulletins from Microsoft and Adobe. For the purpose of brevity and simplicity, we chose to perform our experiments on a modified dataset consisting of 105 vulnerability descriptions taken from the archived Joshi corpus. The modified dataset (will also be referred to as the Joshi Corpus/Dataset from here on) bears the following distribution of source counts: 72 from the National Vulnerability Database, 9 blog entries and 24 security bulletins from Microsoft (MS) and Adobe. This distribution (relative) is similar to the one documented in [13] and hence allows us to compare the accuracy levels obtained in this thesis with those obtained by Joshi et al. [13]. The Joshi Corpus consists of a total of 618 tagged concept instances, the distribution of which is shown in Table 5.2.

[12] `http://ebiquity.umbc.edu/resource/html/id/355`

Our Corpus

Because the Joshi Corpus does not completely encompass all the PAG concepts, we generated a separate corpus with labelled PAG concepts. Unfortunately we could not accumulate a large group of cyber-security proficient students to annotate this corpus. Three students from the Computer Science Department at Colorado State University were asked to manually annotate 5 vulnerability description from each of the three vulnerability databases: National Vulnerability Database, ISS-Xforce and Securityfocus. A total of 15 vulnerability descriptions were annotated. At first, each of the three students tagged the dataset manually. The three generated datasets were referred to as *Student1*, *Student2* and *Student3*. In order to gain increased assurance about the correctness of the tags, we systematically combined selected pairs of datasets to generate two new datasets namely, *Student1-Student2* and *Student1-Student3*. Datasets *Student1-Student2* and *Student1-Student3* consisted of a total of 78 and 69 tokens respectively and their individual distributions are shown in Table 5.3.

5.1.2 Evaluation Metrics

In order to compare the accuracy levels of our work with that obtained by Joshi et al., we decided to model the evaluation procedure on the one used in [13]. This meant that the accuracy of the extraction process was to be expressed in terms of the gold standard evaluations metrics: Precision and Recall [15]. The metrics can be calculated using the following formulas:

- precision = $TruePositives/(TruePositives + FalsePositives)$
- recall = $TruePositives/(TruePositives + FalseNegative)$

In the above mentioned formulas *True Positives (TP)* are the instances of PAG concepts which were correctly extracted, *False Positives (FP)* are the instances of PAG concepts which were incorrectly extracted and *False Negatives (FN)* are the instances of PAG concepts which were incorrectly rejected i.e were not extracted in spite of being present in the textual vulnerability description. *Precision* thus quantifies the ability of our approach to extract correct instances of PAG

concepts, whereas *Recall* quantifies the ability of our approach to extract all instances of PAG concepts present in the vulnerability description.

In course of the evaluation process, we compare various instances of obtained values for these metrics using the Wilcoxon Signed Rank Test. This is a nonparametric test used to compare paired data from a pair of samples without the assumption of observed normality in the two samples. Paired data refers to pairs of observations of the same variable recorded under different experimental conditions. For the purpose of this thesis we compare precision and recall values obtained from different experiments at a confidence level of α. In all of the tests conducted we use the following three p-value statistics:

- $p_{=}$: Denotes the p-value obtained after performing a two-tailed Wilcoxon Signed Rank Test. A $p_{=}$ value of $< \alpha$ denotes rejection of the null hypothesis that the median difference between the pair of observations is 0. In other words, we reject the null hypothesis in favor of the alternative hypothesis: "there is significant difference between the two samples".

- $p_{<}$: Denotes the p-value obtained after performing a one-tailed Wilcoxon Signed Rank Test. A $p_{<}$ value of $< \alpha$ denotes rejection of the null hypothesis that the median difference between the pair of observations is $>= 0$. In other words, we reject the null hypothesis in favor of the alternative hypothesis: "the first sample is less significant".

- $p_{>}$: Denotes the p-value obtained after performing a one-tailed Wilcoxon Signed Rank Test. A $p_{>}$ value of $< \alpha$ denotes rejection of the null hypothesis that the median difference between the pair of observations is $=< 0$. In other words, we reject the null hypothesis in favor of the alternative hypothesis: "the first sample is significant".

Here $\alpha = (1 - \ chosen\ confidence\ interval)$ and due to lack of adequate sample points we choose a confidence interval of 90%.

Table 5.4: Results for Experiments Run on the Joshi Corpus

Sources	PAG Concepts	TP	FP	FN	Precision%	Recall%
NVD	Software Names and Versions	79	21	18	79	81.44
	Modifiers	66	12	28	84.62	70.21
	Attacker Actions	27	19	22	58.70	55.10
	Post-Conditions	37	16	8	69.81	82.22
Adobe Security Bulletins	Software Names and Versions	95	11	50	89.62	65.52
	Modifiers	0	0	0	N/A	N/A
	Attacker Actions	1	1	0	50	100
	Post-Conditions	11	0	0	100	100
MS Security Bulletins	Software Names and Versions	79	38	11	67.52	87.78
	Modifiers	0	0	0	N/A	N/A
	Attacker Actions	7	23	2	23.33	77.79
	Post-Conditions	1	30	17	3.23	5.56
Security Blogs	Software Names and Versions	28	18	16	60.87	63.64
	Modifiers	0	0	2	N/A	0
	Attacker Actions	6	12	4	33.33	60
	Post-Conditions	2	10	1	16.67	66.67
Cumulative	Software Names and Versions	281	88	95	76.15	74.73
	Modifiers	66	12	30	84.62	68.75
	Attacker Actions	41	55	28	42.71	59.42
	Post-Conditions	51	56	26	47.66	66.23
Joshi [13]	Software Names and Versions	2372	258	306	90.19	88.57
	Modifiers	320	79	147	80.20	68.52
	Attacker Actions	185	94	177	66.31	51.10
	Post-Conditions	299	123	135	70.85	68.89

5.2 Evaluating on the Joshi Corpus

Table 5.4 shows the results obtained after our approach was employed to extract PAG concepts from the vulnerability descriptions in the Joshi Corpus. Figure 5.1 provides a pictorial overview of Table 5.4. Different colors (shown in the legends) have been used in Figure 5.1 to differentiate between the obtained concepts. Dashed lines are used to show the precision and recall values obtained by Joshi et al. [13]. The same values are shown in the last four rows of Table 5.4 under the *Sources* label “Joshi [13]". The “Cumulative" results obtained from the experiments are shown using solid lines in Fig. 5.1. It can be seen that, cumulatively, our approach performed better than Joshi’s in identifying *Modifiers*. Although most of the other precision and recall values were comparable, precision scores for identifying *Attacker Actions* and *Post-Conditions* were significantly low. This is because, the number of false positives (as seen in the “FP" column from Table 5.4)

Figure 5.1: Comparative Evaluation of Precision and Recall Scores [Joshi Corpus]

were significantly high. The most obvious reason for this is that our approach identified potential “User Actions" as “Attacker Actions". The root cause for this issue can be traced back to non-fulfillment of heuristics 13 and 14 where either exact actor attributes could not be determined or SentiWordnet (refer to section 3.4) misclassified benign attributes as malicious. Another cause for the low precision scores was incorrect classification of *Post-Conditions* as *Attacker Actions* and vice-versa. This issue was also observed by Joshi et al. [13] in their analysis.

The experiments were carried out individually on each source of vulnerability description, as shown in the *Sources* column of Table 5.4. The results show that our approach to extract *Software Names and Versions* exhibited similar performance across all the vulnerability databases in the Joshi Corpus. This can also be seen in Figure 5.1. As seen in Table 5.2, tagged instances of *Modifiers* were missing in the vulnerability descriptions obtained from Adobe and Microsoft Security Bulletins, thus leading corresponding “N/A" values in the Table 5.4. In general, it was observed that the extraction process performed better on vulnerability descriptions obtained from

Table 5.5: Results for Experiments Run on Our Corpus

		Student1-Student2					Student1-Student2					Average	
Sources	**Concept**	**TP**	**FP**	**FN**	**Precision%**	**Recall%**	**TP**	**FP**	**FN**	**Precision%**	**Recall%**	**Precision%**	**Recall%**
NVD	**Software**	7	2	0	77.78	100	9	0	0	100	100	88.89	100
	Version	5	3	1	62.5	83.33	4	3	1	57.14	80	59.82	81.67
	Modifier	2	2	1	50	66.67	1	1	1	50	50	50	58.33
	User Action	0	2	0	0	N/A	0	2	0	0	N/A	0	N/A
	Attacker Action	3	1	1	75	75	3	1	1	75	75	75	75
	Post-Condition	4	0	2	100	66.67	4	0	1	100	80	100	73.33
Iss-Xforce	**Software**	3	1	2	75	60	3	1	2	75	60	75	60
	Version	0	0	0	N/A	N/A	0	0	0	N/A	N/A	N/A	N/A
	Modifier	0	0	0	N/A	N/A	0	0	0	N/A	N/A	N/A	N/A
	User Action	2	0	2	100	50	2	0	0	100	100	100	75
	Attacker Action	3	10	1	23.08	75	3	10	0	23.08	100	23.08	87.5
	Post-Condition	9	10	0	47.37	100	11	7	0	61.11	100	54.24	100
SecurityFocus	**Software**	7	0	0	100	100	7	0	0	100	100	100	100
	Version	2	0	0	100	100	2	0	0	100	100	100	100
	Modifier	1	0	0	100	100	1	0	0	100	100	100	100
	User Action	0	1	0	0	N/A	0	1	0	0	N/A	0	N/A
	Attacker Action	2	3	0	40	100	0	5	0	0	N/A	20	49.5
	Post-Condition	17	1	1	94.44	94.44	14	4	0	77.78	100	86.11	97.22
Cumulative	**Software**	17	3	2	85	89.47	19	1	2	95	90.48	90	89.97
	Version	7	3	1	70	87.5	6	3	1	66.67	85.71	68.33	86.61
	Modifier	3	2	1	60	75	2	1	1	66.67	66.67	63.33	70.83
	User Action	2	3	2	40	50	2	3	0	40	100	40	75
	Attacker Action	8	14	2	36.36	80	6	16	1	27.27	85.71	31.82	82.86
	Post-Condition	30	11	3	73.17	90.91	29	11	1	72.5	96.67	72.84	93.79

NVD. This was again understandable since most of the heuristics were composed by observing semantic patterns in vulnerability descriptions published by NVD.

5.3 Evaluating on Our Corpus

The results shown in Table 5.5 are similar to those obtained in Table 5.4. The recall values obtained from the experiments performed on Our Corpus were in general better than those obtained from the experiments performed on the Joshi Corpus. This showed that our approach could extract a larger chunk of PAG concepts embedded in the vulnerability descriptions obtained from Our Corpus. However, precision values were still visibly low (refer to Figure 5.2). Figure 5.2 also includes the Joshi Metrics (Table 5.4). This acts as a reference for comparing the results obtained from the two corpora. The primary intention behind introducing Our Corpus was to evaluate the performance of our approach in identifying *User Actions*. However, only a few instances on *User Actions* were observed in the tagged dataset. The “Average" column in Table 5.5 shows the mean *User Action* precision and recall scores as obtained from the “Student1-Student2" and “Student1-Student3" datasets. Although the recall value for this concept identification was 75%, the precision

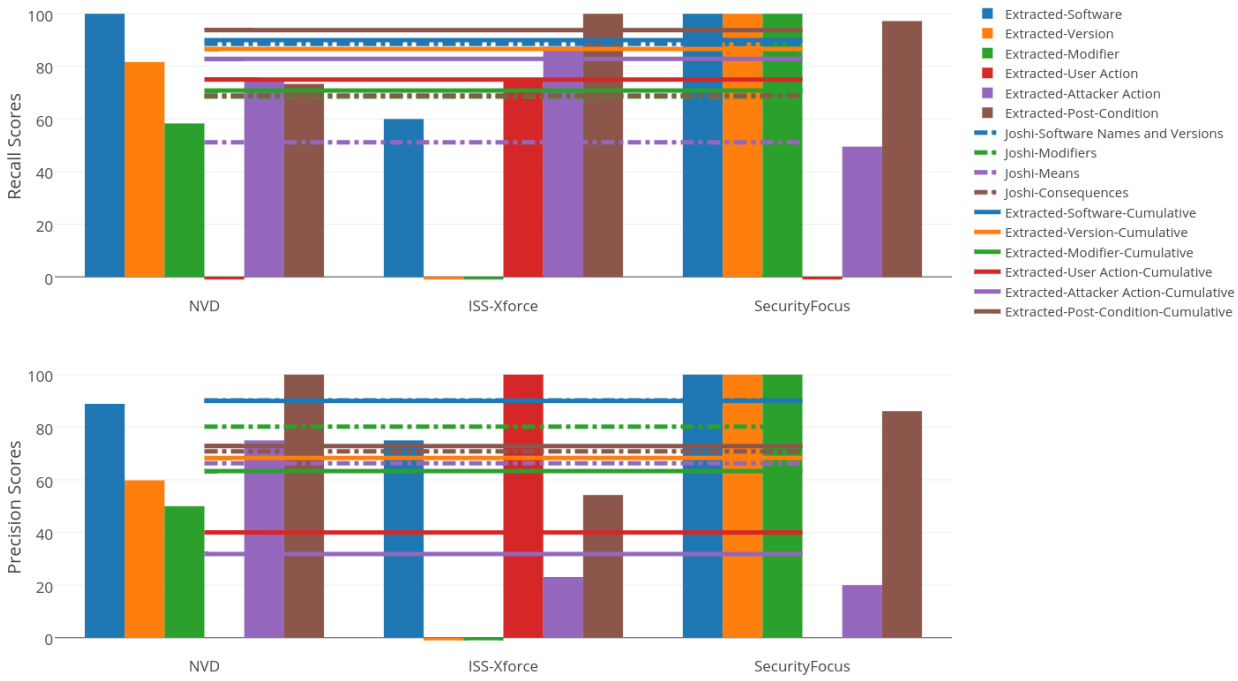


Figure 5.2: Comparative Evaluation of Precision and Recall Scores [Our Corpus]

was observed at 40%. The cause for this phenomenon can be attributed to the exactly same issue as identified while evaluating on the Joshi Corpus. In most cases, heuristic 13 was not successful in assigning correct attributes to identified actors. In other case, SentiWordnet (refer to section 3.4) misclassified benign attributes as malicious. Versions and modifiers also exhibited considerably less precision scores. This is because, heuristics 7 and 8 generated false *Software-Version* links thereby increasing the number of false positive *Version*, and consequently *Modifier*, assignments.

5.4 Effectiveness of the Heuristics

In this section we make an attempt to evaluate the effectiveness of our heuristics across different vulnerability databases. However, before beginning the evaluation process we must note that thorough analysis of vulnerability descriptions obtained from NVD, Securityfocus, and ISS-XForce reveal that they follow general semantic patterns. This does not imply that the patterns are consistent across the three domains, but within these domains these patterns are actively found. However, security bulletins published by Adobe and Microsoft and security blogs exhibit a more

free flowing style of writing [13] and hence do not exhibit commonly occurring semantics. It can thus be observed that the Joshi Corpus consists of a more diverse set of sources in the form of NVD, security bulletins and blogs, while Our Corpus consists of sources which publish more semantically coherent information. Consequently, there is a greater chance that heuristics created from observing NVD information are effective for extracting security related concepts from ISS-Xforce and Securityfocus, but not across security bulletins and blogs. To verify this we perform the following three steps:

5.4.1 Step 1

We need to first ascertain that no bias was introduced in the two results owing to conceptual differences in the datasets. If bias is found, no further comparisons would be possible within results obtained from these two datasets. To verify the absence of any bias, we first define the following hypothesis:

- H_o: Our approach performs significantly better on one Corpora. Rejected if $p\_H_o = (p_= > .1, p_< > .1, p_> > .1)$
- H_a: Our approach performs equitably on the two corpora.

NVD was the common source used in both test datasets. As a result, to evaluate our hypothesis, we perform the Wilcoxon signed rank test on precision and recall scores obtained from the experiments on the NVD vulnerability descriptions in both corpora as shown in Tables 5.4 and 5.5. If the obtained p-values (both two-tailed and one-tailed) are $> .1$, we reject the null hypothesis H_o. The results are tabulated in Table 5.6.

Table 5.6: NVD Precision and Recall P-Values [Joshi and Our Corpus]

	Precision	**Recall**	α
$p_=$	0.875	0.625	.1
$p_<$	0.6875	0.8125	.1
$p_>$	0.4375	0.3125	.1

[13] security blogs are most often updated by regular users

Table 5.7: NVD vs Bulletins and Blogs Precision and Recall Samples for Wilcoxon Test [Joshi Corpus]

Precision Scores		**Recall Scores**	
NVD	**Bulletins and Blogs**	**NVD**	**Bulletins and Blogs**
79	89.622641509434	81.4432989690722	65.5172413793104
79	67.5213675213675	81.4432989690722	87.7777777777778
79	60.8695652173913	81.4432989690722	63.6363636363636
58.695652173913	50	55.1020408163265	100
58.695652173913	23.3333333333333	55.1020408163265	77.7777777777778
58.695652173913	33.3333333333333	55.1020408163265	60
69.811320754717	100	82.2222222222222	100

The results in Table 5.6 show there is not enough evidence to conclude there was any bias introduced by conceptual differences between the two datasets. Interestingly, this also shows that our approach can be used in other cyber-security related concept extraction frameworks.

5.4.2 Step 2

With the above observation, we define the next pair of hypothesis as follows:

- H_o: Heuristics generated by observing semantic patterns in vulnerability descriptions published by NVD were effective in extracting PAG concepts from other sources like security bulletins and blogs.

  Rejected if $p\_H_o = (p_= < .1, p_< > .1, p_> < .1)$

- H_a: Heuristics generated by observing semantic patterns in vulnerability descriptions published by NVD cannot be used to extract PAG concepts from other sources like security bulletins and blogs.

We then perform another signed rank sum test on precision and recall scores obtained from experiments performed on the Joshi Corpus. The sample points are obtained from Table 5.4 (except the “N/A" labeled concepts/rows) and are shown in Table 5.7. It can be noted that each NVD precision/recall score is paired with a corresponding precision/recall score for bulletins and blogs from Table 5.4. The obtained p-values are tabulated below in Table 5.8.

Table 5.8: NVD vs Bulletins and Blogs Precision and Recall P-Values [Joshi Corpus]

	Precision	**Recall**	α
$p_=$	0.0977	1.00	.1
$p_<$	0.96	0.5	.1
$p_>$	0.0488	0.54	.1

Table 5.9: NVD vs Other Semi-Structured Precision and Recall Samples for Wilcoxon Test [Our Corpus]

Precision		**Recall**	
NVD	**Other Semi Structured**	**NVD**	**Other Semi Structured**
88.8888888888889	75	100	60
88.8888888888889	100	100	100
59.8214285714286	100	81.6666666666667	100
50	100	58.3333333333333	100
0	100	75	87.5
0	0	75	49.5
75	23.0769230769231	73.3333333333333	100
75	20	73.3333333333333	97.2222222222222
100	54.2397660818714		
100	86.1111111111111		

From Table 5.8 it can be observed that at 90% confidence interval there is enough evidence that the NVD precision scores are better than those of “Bulletins and Blogs", but the same cannot be said about the recall scores. This means that, at least precision wise, we can reject the null hypothesis H_o in favor of the alternative hypothesis that heuristics generated from NVD published vulnerability descriptions did not perform as well on sources like security bulletins, blogs.

5.4.3 Step 3

Table 5.10: NVD vs Bulletins and Blogs Precision and Recall P-Values [Joshi Corpus]

	Precision	**Recall**	α
$p_=$	0.7668	0.499	.1
$p_<$	0.3834	0.2495	.1
$p_>$	0.6166	0.7505	.1

We now attempt to perform a similar Wilcoxon test on Our Corpora. For this purpose, we define the following hypothesis:

- H_o: Heuristics generated by observing semantic patterns in vulnerability descriptions published by NVD were not effective in extracting PAG concepts from sources like ISS-XForce and Securityfocus. Rejected if $p\_H_o = (p_= > .1, p_< > .1, p_> > .1)$

- H_a: Heuristics generated by observing semantic patterns in vulnerability descriptions published by NVD were effective in extracting PAG concepts from sources like ISS-XForce and Securityfocus.

Table 5.9 shows the sample points used for the Wilcoxon signed rank test. The sample generation strategy is similar to the one described in the previous step. These samples are generated from Table 5.5. In Table 5.10 it can be seen that $p_= > .1$, $p_< > .1$, $p_> > .1$. Under these observations we can reject the null hypothesis H_o and thus conclude that the heuristics extracted from vulnerability descriptions published by NVD were effective in extracting information from other databases which publish information with some semantic coherence.

In general, it can thus be concluded, with some level of confidence, that heuristics extracted from vulnerability descriptions published by NVD were effective in extracting information from only those vulnerability databases which publish semantically coherent information.

Chapter 6

Conclusion and Future work

A Personalized Attack Graph (PAG) [24, 25, 32] is a graphical representation of the interaction between vulnerabilities existing on a system and actions performed by users and attacker which lead to a successful compromise of the system. In order to realize this representation, one needs to encode propositions in the form of node labels. Textual representations of such propositions, embedded in vulnerability descriptions, can be found in online vulnerability databases like the National Vulnerability Database (NVD) and ISS-XForce and various security bulletins and blogs. In order to realize the PAG, downloaded vulnerability descriptions need to be parsed into four distinct classes namely, *Attacker Actions*, *User Actions*, *Software and Versions* and *Post-Conditions*. These classes are cumulatively referred to as the PAG concepts. However, vulnerability descriptions are most often written and updated by human beings and are hence mostly unstructured in nature. Extracting PAG concepts from unstructured vulnerability descriptions is a non-trivial task and to perform it in an automated fashion is even more challenging.

In this thesis, we have shown two separate approaches: an approach to extract *Software and Versions* and a second approach to extract *Attacker Actions*, *User Actions* and *Post-Condition*. The core elements of our approaches are heuristics developed by thoroughly observing semantic patterns in vulnerability descriptions obtained from the National Vulnerability Database. In general, to the best of our knowledge, ours is the first approach which does not rely on supervised machine learning techniques to extract cyber security related concepts.

We evaluated our approaches using gold standard metrics (Precision and Recall) and compared the results with those obtained in one of the most prominent works [13] in the field on cyber security related concept extraction. Our results were comparable with those obtained in [13] which is a machine learning based solution and requires explicit training data to function. Finally, we performed non-parametric significance tests to demonstrate the viability of our rules for PAG related concept extraction from across various types of databases, including security bulletins and

blogs. In the same context, we also showed that our approach can be suitable to be used across other concept extraction frameworks. Moreover, since this is an unsupervised approach, it could easily be tested on any concept tagged dataset.

Extracting PAG concepts maybe the most important task, but it is certainly not the only task in building the PAG. In this thesis, we proposed an approach to parsing textual information from popular vulnerability databases. However, there are a plethora of such vulnerability databases and till date, there is no clear consensus as to which sources provide the best quality and most timely information. Furthermore, the underlying HTML tree structure of vulnerability database websites can be diverse and complex. Thus, extracting vulnerability descriptions from vulnerability database websites is another aspect we plan to look into, in the future. Finally, once the PAG concepts are extracted from vulnerability descriptions, they need to be interlinked to form the Personalized Attack Graph. This requires finding semantic linkages between various concepts and can be a highly challenging task.

Bibliography

[1] C. L. Anderson and R. Agarwal. Practicing safe computing: A multimedia empirical examination of home computer user security behavioral intentions. *MIS Q.*, 34(3):613–643, Sept. 2010.

[2] S. Baccianella, A. Esuli, and F. Sebastiani. Sentiwordnet 3.0: An enhanced lexical resource for sentiment analysis and opinion mining. In N. C. C. Chair), K. Choukri, B. Maegaard, J. Mariani, J. Odijk, S. Piperidis, M. Rosner, and D. Tapias, editors, *Proceedings of the Seventh International Conference on Language Resources and Evaluation (LREC'10)*, Valletta, Malta, may 2010. European Language Resources Association (ELRA).

[3] C. Bizer, J. Lehmann, G. Kobilarov, S. Auer, C. Becker, R. Cyganiak, and S. Hellmann. Dbpedia-a crystallization point for the web of data. *Web Semantics: science, services and agents on the world wide web*, 7(3):154–165, 2009.

[4] R. A. Bridges, C. L. Jones, M. D. Iannacone, K. M. Testa, and J. R. Goodall. Automatic labeling for entity extraction in cyber security. *arXiv preprint arXiv:1308.4941*, 2013.

[5] M.-C. de Marneffe, B. MacCartney, and C. D. Manning. Generating typed dependency parses from phrase structure parses. In *Proceedings of the International Conference on Language Resources and Evaluation*, pages 449–454, 2006.

[6] M.-C. de Marneffe and C. D. Manning. Stanford typed dependencies manual, September 2008.

[7] M.-C. de Marneffe and C. D. Manning. The stanford typed dependencies representation. In *Coling 2008: Proceedings of the Workshop on Cross-Framework and Cross-Domain Parser Evaluation*, CrossParser '08, pages 1–8, Stroudsburg, PA, USA, 2008. Association for Computational Linguistics.

[8] R. Dewri, N. Poolsappasit, I. Ray, and D. Whitley. Optimal security hardening using multi-objective optimization on attack tree models of networks. In *Proceedings of the 14th ACM conference on Computer and communications security*, pages 204–213. ACM, 2007.

[9] A. Esuli and F. Sebastiani. Sentiwordnet: A publicly available lexical resource for opinion mining. In *In Proceedings of the 5th Conference on Language Resources and Evaluation*, pages 417–422, 2006.

[10] C. Fellbaum. *WordNet: An Electronic Lexical Database*. Bradford Books, 1998.

[11] T. Finin and Z. Syed. Creating and exploiting a web of semantic data. In *ICAART (1)*, pages 7–18, 2010.

[12] S. Jha, O. Sheyner, and J. Wing. Two formal analyses of attack graphs. In *Computer Security Foundations Workshop, 2002. Proceedings. 15th IEEE*, pages 49–63. IEEE, 2002.

[13] A. Joshi, L. Ravendar, T. Finin, and A. Joshi. Extracting cyberscurity related linked data from text. In *Proceedings of the 7th IEEE International Conference on Semantic Computing*, pages 252–259, 2013.

[14] D. Klein and C. D. Manning. Accurate unlexicalized parsing. In *Proceedings of the 41st Annual Meeting on Association for Computational Linguistics*, pages 423–430, Stroudsburg, PA, USA, 2003.

[15] J. Makhoul, F. Kubala, R. Schwartz, R. Weischedel, et al. Performance measures for information extraction. In *Proceedings of DARPA broadcast news workshop*, pages 249–252, 1999.

[16] M. P. Marcus, M. A. Marcinkiewicz, and B. Santorini. Building a large annotated corpus of english: The penn treebank. *Comput. Linguist.*, 19(2):313–330, June 1993.

[17] N. McNeil, R. A. Bridges, M. D. Iannacone, B. Czejdo, N. Perez, and J. R. Goodall. Pace: Pattern accurate computationally efficient bootstrapping for timely discovery of cybersecurity concepts. In *Machine Learning and Applications (ICMLA), 2013 12th International Conference on*, volume 2, pages 60–65. IEEE, 2013.

[18] G. A. Miller. Wordnet: A lexical database for english. *Commun. ACM*, 38(11):39–41, Nov. 1995.

[19] V. Mulwad, W. Li, A. Joshi, T. Finin, and K. Viswanathan. Extracting information about security vulnerabilities from web text. In *Web Intelligence and Intelligent Agent Technology (WI-IAT), 2011 IEEE/WIC/ACM International Conference on*, volume 3, pages 257–260. IEEE, 2011.

[20] D. Nadeau and S. Sekine. A survey of named entity recognition and classification. *Lingvisticae Investigationes*, 30(1):3–26, 2007.

[21] N. Poolsapassit and I. Ray. Investigating computer attacks using attack trees. In *Advances in digital forensics III*, pages 331–343. Springer, 2007.

[22] N. Poolsappasit, R. Dewri, and I. Ray. Dynamic security risk management using bayesian attack graphs. *Dependable and Secure Computing, IEEE Transactions on*, 9(1):61–74, 2012.

[23] I. Ray and N. Poolsapassit. Using attack trees to identify malicious attacks from authorized insiders. In *Computer Security–ESORICS 2005*, pages 231–246. Springer, 2005.

[24] M. Roberts, A. Howe, I. Ray, M. Urbanska, Z. S. Byrne, and J. M. Weidert. Personalized vulnerability analysis through automated planning. In *Working Notes of IJCAI 2011, Workshop Security and Artificial Intelligence (SecArt-11),*, 04 2011.

[25] M. Roberts, A. E. Howe, I. Ray, and M. Urbanska. Using planning for a personalized security agent. In *Workshop on Problem Solving using Classical Planners in Working Notes of the 26th AAAI Conference on Artificial Intelligence*, Toronto, Ontario, Canada, July 2012.

[26] S. Roschke, F. Cheng, and C. Meinel. Using vulnerability information and attack graphs for intrusion detection. In *Information Assurance and Security (IAS), 2010 Sixth International Conference on*, pages 68–73. IEEE, 2010.

[27] S. Roschke, F. Cheng, R. Schuppenies, and C. Meinel. Towards unifying vulnerability information for attack graph construction. In *Information security*, pages 218–233. Springer, 2009.

[28] B. Santorini. Part-of-speech tagging guidelines for the penn treebank project (3rd revision). 1990.

[29] Secunia. The secunia vulnerability review 2014. http://secunia.com/resources/reports/vr2015/, 2014.

[30] O. Sheyner, J. Haines, S. Jha, R. Lippmann, and J. M. Wing. Automated generation and analysis of attack graphs. In *Security and privacy, 2002. Proceedings. 2002 IEEE Symposium on*, pages 273–284. IEEE, 2002.

[31] M. Urbanska et al. Structuring a vulnerability description for comprehensive single system security analysis. In *Rocky Mountain Celebration of Women in Computing*, Fort Collins, CO, USA, 2012.

[32] M. Urbanska et al. Accepting the inevitable: Factoring the user into home computer security. In *Proceedings of the Third ACM Conference on Data and Application Security and Privacy*, San Antonio, TX, USA, February 2013.

[33] H. M. Wallach. Conditional random fields: An introduction. *Technical Reports (CIS)*, page 22, 2004.

[34] L. Wang, M. Albanese, and S. Jajodia. Attack graph and network hardening. In *Network Hardening*, SpringerBriefs in Computer Science, pages 15–22. Springer International Publishing, 2014.