Towards the Detection of Inconsistencies in Public Security Vulnerability Reports

Ying Dong[1,2] [*], Wenbo Guo[2,4], Yueqi Chen[2,4],
Xinyu Xing[2,4], Yuqing Zhang[1], and Gang Wang[3]

[1]*School of Computer Science and Technology, University of Chinese Academy of Sciences, China*
[2]*College of Information Sciences and Technology, The Pennsylvania State University, USA*
[3]*Department of Computer Science, Virginia Tech, USA*
[4]*JD Security Research Center, USA*
dongying115@mails.ucas.ac.cn, {wzg13, yxc431, xxing}@ist.psu.edu
zhangyq@ucas.ac.cn, gangwang@vt.edu

Abstract

Public vulnerability databases such as the Common Vulnerabilities and Exposures (CVE) and the National Vulnerability Database (NVD) have achieved great success in promoting vulnerability disclosure and mitigation. While these databases have accumulated massive data, there is a growing concern for their information quality and consistency.

In this paper, we propose an automated system `VIEM` to detect inconsistent information between the fully *standardized* NVD database and the *unstructured* CVE descriptions and their referenced vulnerability reports. `VIEM` allows us, for the first time, to quantify the information consistency at a massive scale, and provides the needed tool for the community to keep the CVE/NVD databases up-to-date. `VIEM` is developed to extract vulnerable software names and vulnerable versions from unstructured text. We introduce customized designs to deep-learning-based named entity recognition (NER) and relation extraction (RE) so that `VIEM` can recognize previous unseen software names and versions based on sentence structure and contexts. Ground-truth evaluation shows the system is highly accurate (0.941 precision and 0.993 recall). Using `VIEM`, we examine the information consistency using a large dataset of 78,296 CVE IDs and 70,569 vulnerability reports in the past 20 years. Our result suggests that inconsistent vulnerable software versions are highly prevalent. Only 59.82% of the vulnerability reports/CVE summaries strictly match the standardized NVD entries, and the inconsistency level increases over time. Case studies confirm the erroneous information of NVD that either overclaims or underclaims the vulnerable software versions.

1 Introduction

Security vulnerabilities in computer and networked systems are posing a serious threat to users, organizations, and nations at large. Unmatched vulnerabilities often lead to real-world attacks with examples ranging from WannaCry ransomware that shut down hundreds of thousands of machines in hospitals and schools [20] to the Equifax data breach that affected half of America's population [21].

To these ends, a strong *community effort* has been established to find and patch vulnerabilities before they are exploited by attackers. The Common Vulnerabilities and Exposures (CVE) program [4] and the National Vulnerability Database (NVD) [11] are among the most influential forces. CVE is a global list/database that indexes publicly known vulnerabilities by harnessing the "the power of the crowd". Anyone on the Internet (security vendors, developers and researchers) can share the vulnerabilities they found on CVE. NVD is a more standardized database established by the U.S. government (*i.e.,* NIST). NVD receives data feeds from the CVE website and perform analysis to assign common vulnerability severity scores (CVSS) and other pertinent metadata [18]. More importantly, NVD standardizes the data format so that algorithms can directly process their data [12]. Both CVE and NVD play an important role in guiding the vulnerability mitigation. So far, over 100,000 vulnerabilities were indexed, and the CVE/NVD data stream has been integrated with hundreds of security vendors all over the world [10].

While the vulnerability databases are accumulating massive data, there is also a growing concern about the *information quality* [28, 42, 44]. More specifically, the information listed on CVE/NVD can be incomplete or outdated, making it challenging for researchers to reproduce the vulnerability [42]. Even worse, certain CVE entries contain erroneous information which may cause major delays in developing and deploying patches. In practice, industrial systems often use legacy software for a long time due to the high cost of an update. When a relevant vulnerability is disclosed, system administrators usually look up to vulnerability databases to determine whether their software (and which versions) need to be patched. In addition, CVE/NVD are serving as a key information source for security companies to assess the secu-

[*]This work was done when Ying Dong studied at the Pennsylvania State University.

rity level of their customers. Misinformation on CVE/NVD could have left critical systems unpatched.

In this paper, we propose a novel system to automatically detect inconsistent information between the fully *standardized* NVD database and the *unstructured* CVE descriptions and their referenced vulnerability reports. Our system `VIEM` allows us, for the first time, to quantify the information consistency at a massive scale. Our study focuses on *vulnerable software versions*, which is one of the most important pieces of information for vulnerability reproduction and vulnerability patching. We face three main technical challenges to build `VIEM`. First, due to the high diversity of software names and versions, it is difficult to build dictionaries or regular expressions [30, 48, 59] to achieve high precision and recall. Second, the unstructured text of vulnerability reports and summaries often contain code, and the unique writing styles are difficult to handle by traditional natural language processing tools [22, 49, 50]. Third, we need to extract "*vulnerable*" software names and their versions, and effectively exclude the distracting items (*i.e.*, non-vulnerable versions).

Our System. To address these challenges, we build `VIEM` (short for Vulnerability Information Extraction Model) with a Named Entity Recognition (NER) model and a Relation Extraction (RE) model. The goal is to learn the patterns and indicators from the sentence structures to recognize the vulnerable software names/versions. Using "contexts" information, our model can capture previously unseen software names and is generally applicable to different vulnerability types. More specifically, the `NER` model is a recurrent deep neural network [36, 56] which pinpoints the relevant entities. It utilizes word and character embeddings to encode the text and then leverages a sequence-to-sequence bi-directional GRU (Gated Recurrent Unit) to locate the names and versions of the vulnerable software. The `RE` model is trained to analyze the relationships between the extracted entities to pair the vulnerable software names and their versions together. Finally, to generalize our model to handle different types of vulnerabilities, we introduce a transfer learning step to minimize the manual annotation efforts.

Evaluation and Measurement. We collect a large dataset from the CVE and NVD databases and 5 highly popular vulnerability reporting websites. The dataset covers 78,296 CVE IDs and 70,569 vulnerability reports across all 13 vulnerability categories in the past 20 years. For evaluation, we manually annotated a sample of 5,193 CVE IDs as the ground-truth. We show that our system is highly accurate with a precision of 0.941 and a recall of 0.993. In addition, our model is generalizable to all 13 vulnerability categories.

To detect inconsistencies in practice, we apply `VIEM` to the full dataset. We use NVD as the standard (since it is the last hop of the information flow), and examine the inconsistencies between NVD entries and the CVE entries/external sources. We have a number of key findings. First, the inconsistency level is very high. Only 59.82% of the external reports/CVE summaries have exactly the same vulnerable software versions as those of the standardized NVD entries. It's almost equally common for an NVD entry to "overclaim" or "underclaim" the vulnerable versions. Second, we measure the consistency level between NVD and other sources and discover the inconsistency level increased over the past 20 years (but started to decrease since 2016). Finally, we select a small set of CVE IDs with highly inconsistent information to manually verify the vulnerabilities (including 185 software versions). We confirm real cases where the official NVD/CVE entries and/or the external reports falsely included non-vulnerable versions and missed truly vulnerable versions. Such information could affect systems that depend on NVD/CVE to make critical decisions.

Applications. Detecting information inconsistency is the *first step* to updating the outdated entries and mitigating errors in the NVD and CVE databases. There was no existing tool that could automatically extract vulnerable software names and versions from unstructured reports before. A key contribution of `VIEM` is to enable the possibility to continuously monitor different vulnerability reporting websites and periodically generate a "diff" from the CVE/NVD entries. This can benefit the community in various ways. For employees of CVE/NVD, `VIEM` can notify them whenever a new vulnerable version is discovered for an existing vulnerability (to accelerate the testing and entry updates). For security companies, `VIEM` can help to pinpoint the potentially vulnerable versions of their customers' software to drive more proactive testing and patching. For software users and system administrators, the "diff" will help them to make more informed decisions on software updating. To facilitate future research and application development, we released our labeled dataset and the source code of VIEM[1].

In summary, our paper makes three key contributions.

- *First*, we design and develop a novel system `VIEM` to extract vulnerable software names and versions from unstructured vulnerability reports.
- *Second*, using a large ground-truth dataset, we show that our system is highly accurate and generalizes well to different vulnerability types.
- *Third*, by applying `VIEM`, we perform the first large-scale measurement of the information consistency for CVE and NVD. The generated "diff" is helpful to drive more proactive vulnerability testing and information curation.

2 Background and Challenges

We first introduce the background of security vulnerability reporting, and describe the technical challenges of our work.

[1]https://github.com/pinkymm/inconsistency_detection

2.1 Vulnerability Reporting

CVE. When people identify a new vulnerability, they can request a unique CVE-ID number from one of the CVE Numbering Authorities (CNAs) [5]. The *MITRE Corporation* is the editor and the primary CNA [19]. CNA will then do research on the vulnerability to determine the details and check if the vulnerability has been previously reported. If the vulnerability is indeed new, then a CVE ID will be assigned and the corresponding vulnerability information will be publicly released through the CVE list [4, 9].

The CVE list [4] is maintained by MITRE as a website on which the CVE team publishes a summary for each of the reported vulnerabilities. As specified in [8], when writing a CVE summary, the CVE team will analyze (public) third-party vulnerability reports and then include details in their description such as the name of the affected software, the vulnerable software versions, the vulnerability type, and the conditions/requirements to exploit the vulnerability.

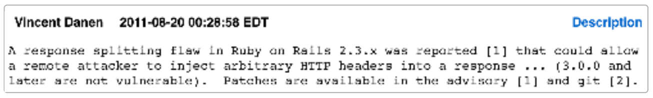
In addition to the summary, each CVE entry contains a list of external references. The external references are links to third-party technical reports or blog/forum posts that provide the needed information for the CVE team to craft the official vulnerability description [1]. The information on CVE can help software vendors and system administrators to pinpoint the versions of the vulnerable software, assess their risk level, and perform remediation accordingly.

NVD. NVD (National Vulnerability Database) is maintained by a different organization (*i.e.,* NIST) from that of CVE [3]. NVD is built fully synchronized with the CVE list. The goal is that any updates to CVE will appear immediately in NVD. After a new CVE ID appears on the CVE list, the NIST NVD team will first perform analysis to add enhanced information such as the severity score before creating the NVD entries [18].

Compared with CVE, NVD provides two additional features. First, NVD data entries are structured. The NIST NVD team would convert the unstructured CVE information into structured JSON or XML, where information fields such as vulnerable software names and versions are formatted and standardized based on the Common Weakness Enumeration Specification (CWE) [12]. Second, data entries are continuously updated. The information in NVD may be updated (manually) after the initial vulnerability reporting. For example, as time goes by, new vulnerable software versions may be discovered by NIST employees or outsiders, which will be added to the existing NVD entries [17].

2.2 Technical Challenges

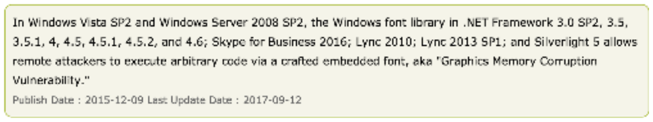
CVE and NVD databases are primarily maintained by *manual efforts*, which leads to a number of important questions. First, given that a vulnerability may be reported and discussed in many different places, how complete is the information (*e.g.*, vulnerable software names and their versions) in the CVE/NVD database? Second, considering the continuous community effort to study a reported vulnerability, how effective is the current manual approach to keep the CVE/NVD entries up-to-date?



Vincent Danen   2011-08-20 00:28:58 EDT   Description

```
A response splitting flaw in Ruby on Rails 2.3.x was reported [1] that could allow
a remote attacker to inject arbitrary HTTP headers into a response ... (3.0.0 and
later are not vulnerable).  Patches are available in the advisory [1] and git [2].
```

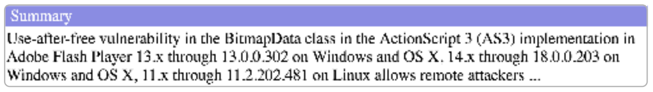
(a) An Openwall report containing the vulnerable versions of software (`2.3.x`) and non-vulnerable versions (`3.0.0 and later`).



In Windows Vista SP2 and Windows Server 2008 SP2, the Windows font library in .NET Framework 3.0 SP2, 3.5, 3.5.1, 4, 4.5, 4.5.1, 4.5.2, and 4.6; Skype for Business 2016; Lync 2010; Lync 2013 SP1; and Silverlight 5 allows remote attackers to execute arbitrary code via a crafted embedded font, aka "Graphics Memory Corruption Vulnerability."

Publish Date : 2015-12-09 Last Update Date : 2017-09-12

(b) A CVE summary enclosing multiple entities pertaining to vulnerable software (the vulnerable component: `Windows font library`; vulnerable software: `.NET Framework`, `Skype for Business`, `Lync`, `Silverlight`; dependent software: `Windows`; software versions tied to these entities).



Summary

Use-after-free vulnerability in the BitmapData class in the ActionScript 3 (AS3) implementation in Adobe Flash Player 13.x through 13.0.0.302 on Windows and OS X, 14.x through 18.0.0.203 on Windows and OS X, 11.x through 11.2.202.481 on Linux allows remote attackers ...

(c) A CVE summary in which the name and versions of vulnerable software are not adjacent.

Figure 1: Examples of vulnerability descriptions and reports.

Our goal is to thoroughly understand the inconsistencies between external vulnerability reporting websites and the CVE/NVD data entries. According to the statistics from [6], the CVE list has archived more than 100,000 distinct CVEs (although certain CVE IDs were merged or withdrawn). Each CVE ID also has 5∼30 external third-party reports. It is infeasible to extract such information manually. The main challenge is to automatically and accurately extract relevant information items from the unstructured reports.

Many existing NLP tools aim to extract relevant information from text (*e.g.*, [22, 49, 50, 53]). However, the unique characteristics of vulnerability reports impose significant challenges, making existing techniques inadequate. ❶ Previously unseen software emerges: the CVE list introduces new vulnerable software frequently, making it difficult to use a pre-defined dictionary to identify the names of all vulnerable software. As such, dictionary-based method is not suitable for this problem (*e.g.*, [30, 48]). ❷ Reports are unstructured: most CVE summaries and vulnerability reports are highly unstructured and thus simple regular-expression-based techniques (*e.g.*, [28, 59]) can be barely effective. ❸ Non-interested entities are prevalent: a vulnerability report usually encloses information about both vulnerable and non-vulnerable versions of software (see Figure 1a). Our goal is to extract "vulnerable" software names and versions while excluding information items related to non-vulnerable software. Techniques that rely on pre-defined rules would hardly work here (*e.g.*, [28, 55, 59]). ❹ Multiple interested entities exist: the vulnerable software mentioned in a report usually refers to multiple entities (see Figure 1b) and the relationships of these entities are determined by the context of the report. This requirement eliminates techniques that lack the capability of handling multiple entities (e.g., [28, 32, 59]). ❺ Vulnerability types are diverse: CVE covers a variety of vulnerability types, and each has its own characteristics in the descriptions. As a result, we cannot simply use techniques designed for certain vulnerability types. For example, a tool used by [59] is designed specifically for kernel memory corruption vulnerabilities. We tested it against our ground-truth dataset and did not receive satisfying results (the recall is below 40%).

3 The Design of `VIEM`

To tackle the challenges mentioned above, we develop an automated tool `VIEM` by combining and customizing a set of state-of-the-art natural language processing (NLP) techniques. In this section, we briefly describe the design of `VIEM` and discuss the reasons behind our design. Then, we elaborate on the NLP techniques that `VIEM` adopts.

3.1 Overview

To pinpoint and pair the entities of our interest, we design `VIEM` to complete three individual tasks.

Named Entity Recognition Model. First, `VIEM` utilizes a state-of-the-art Named Entity Recognition (NER) model [36, 56] to identify the entities of our interest, *i.e.,* the name and versions of the vulnerable software, those of vulnerable components and those of underlying software systems that vulnerable software depends upon (see Figure 1b).

The reasons behind this design are twofold. First, an NER model pinpoints entities based on the structure and semantics of input text, which provides us with the ability to track down software names that have never been observed in the training data. Second, an NER model can learn and distinguish the contexts pertaining to vulnerable and non-vulnerable versions of software, which naturally allows us to eliminate non-vulnerable versions of software and pinpoint only the entities of our interest.

Relation Extraction Model. With the extracted entities, the next task of `VIEM` is to pair identified entities accordingly. As is shown in Figure 2, it is common that software name and version jointly occur in a report. Therefore, one instinctive reaction is to group software name and version nearby, and then deem them as the vulnerable software and version pairs. However, this straightforward approach is not suitable for our problem. As is depicted in Figure 1c, the vulnerable software name is not closely tied to all the vulnerable versions. Merely applying the approach above, we might inevitably miss the versions of the vulnerable software.

To address this issue, `VIEM` first goes through all the possible combinations between versions and software names. Then, it utilizes a Relation Extraction (RE) model [38, 62] to determine the most possible combinations and deems them as the correct pairs of entities. The rationale behind this design is as follows. The original design of an RE model is not for finding correct pairs among entities. Rather, it is responsible for determining the property of a pair of entities. For example, assume that an RE model is trained to assign a pair of entities one of the following three properties – *"born in"*, *"employed by"* and *"capital of"*. Given two pairs of entities $P_1 =$ (*"Steve Jobs"*, *"Apple"*) and $P_2 =$ (*"Steve Jobs"*, "California") in the text *"Steve Jobs was born in California, and was the CEO of Apple."*, an RE model would assign the *"employed by"* to P_1 and *"born in"* properties to P_2.

In our model, each of the possible version-and-software combinations can be treated as an individual pair of entities. Using the idea of the relation extraction model, `VIEM` assigns each pair a property, indicating the truthfulness of the relationship of the corresponding entities. Then, it takes the corresponding property as the true pairing. Take the case in Figure 2 for example, there are 4 entities indicated by 2 software (`Microsoft VBScript` and `Internet Explorer`) and 2 ranges of versions (`5.7 and 5.8` and `9 through 11`). They can be combined in 4 different ways. By treating the combinations as 4 different pairs of entities, we can use an RE model to assign a binary property to each of the combinations. Assuming that the binary property assigned indicates whether the corresponding pair of the entities should be grouped as software and its vulnerable versions, `VIEM` can use the property assignment as the indicator to determine the entity pairing. It should be noted that we represent paired entities in the Common Platform Enumeration (CPE) format [2]. For example, `cpe:/a:google:chrome:3.0.193.2:beta`, where `google` denotes the vendor, `chrome` denotes the product, `3.0.193.2` denotes the version number, and `beta` denotes the software update.

Transfer Learning. Recall that we need to measure vulnerability reports across various vulnerability types. As mentioned in §2.2, the reports of different vulnerability types do not necessarily share the same data distribution. Therefore, it is not feasible to use a single machine learning model to deal with all vulnerability reports, unless we could construct and manually annotate a large training dataset that covers all kinds of vulnerabilities. Unfortunately, there is no such labeled dataset available, and labeling a large dataset involves tremendous human efforts. To address this problem, `VIEM` takes the strategy of transfer learning, which learns the aforementioned NER and RE models using vulnerability reports in one primary category and then transfers their capability into other vulnerability categories. In this way, we can re-

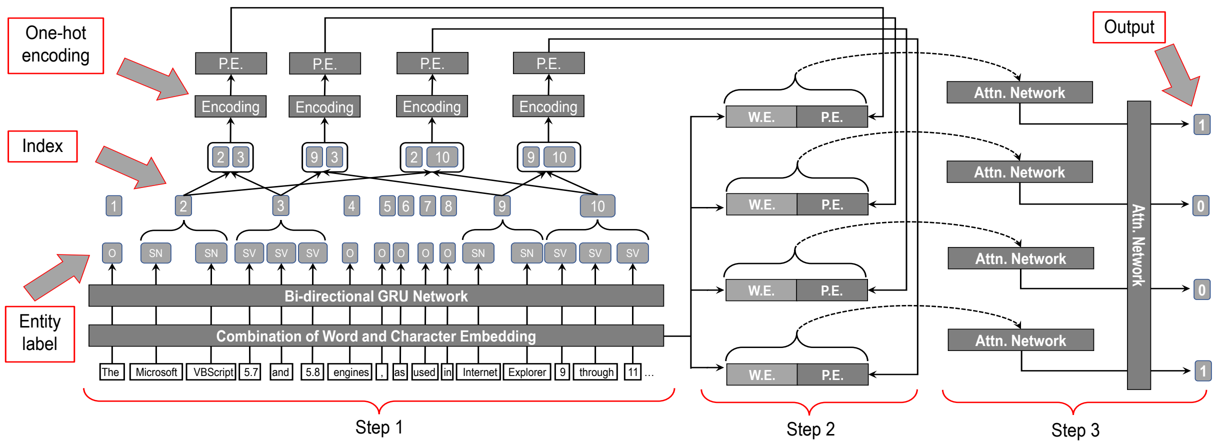


Figure 2: The RE model that pairs the entities of our interests through a three-step procedure. Here, the entities of our interest include software (`Microsoft VBScript`; `Internet Explorer`) as well as version range (`5.7 and 5.8`; `9 through 11`). W.E. and P.E denote word and position embeddings, respectively.

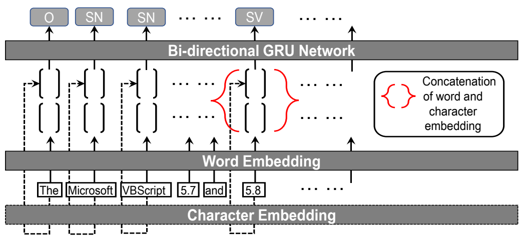

O
SN
SN
SV
Bi-directional GRU Network
Concatenation of word and character embedding
Word Embedding
The
Microsoft
VBScript
5.7
and
5.8
Character Embedding

Figure 3: The NER model which utilizes both word and character embeddings to encode a target text and then a Bi-GRU network to assign entity labels to each word.

duce the efforts involved in data labeling, making `VIEM` effective for arbitrary kinds of vulnerability reports. More details of the transfer learning are discussed in §5.

3.2 Named Entity Recognition Model

We start by extracting named entities (vulnerabile software names and versions) from the text. We develop our system based on a recent NER model [36,56]. On top of that, we integrate a gazetteer to improve its accuracy of extracting vulnerable software names. At the high level, the NER model first encodes a text sequence into a sequence of word vectors using the concatenation of word and character embeddings. This embedding process is necessary since a deep neural network cannot process text directly. Then taking the sequence of the word vectors as the input, the model predicts a label for each of the words in the sequence using a bi-directional Recurrent Neural Network. Below, we introduce the key technical details.

For word and character embeddings, the NER model first utilizes a standard word embedding approach [40] to encode each word as a vector representation. Then, it utilizes a Bi-directional Gated Recurrent United (Bi-GRU) network to perform text encoding at the character level. As shown in Figure 3, the NER model concatenates these two embeddings as a single sequence of vectors and then takes it as the input for another Bi-GRU network.

Different from the aforementioned Bi-GRU network used for text embedding, the second Bi-GRU network is responsible for assigning labels to words. Recall that our task is to identify the entities of our interest which pertain to vulnerable software. As a result, we use the Bi-GRU network to pinpoint the words pertaining to this information. More specifically, we train this Bi-GRU network to assign each word with one of the following labels – ❶ `SV` (software version), ❷ `SN` (software name) and ❸ `O` (others) – indicating the property of that word. It should be noted that we assign the same `SN` label to vulnerable software, vulnerable component and the underlying software that vulnerable software is dependent upon. This is because this work measures version inconsistencies of all software pertaining to a vulnerability[2].

Considering that the NER model may not perfectly track down the name of the vulnerable software, we further construct a gazetteer (*i.e.,* a dictionary consisting of 81,551 software mentioned in [10]) to improve the recognition performance of the NER model. To be specific, we design a heuristic approach to rectify the information that the NER model fails to identify or mistakenly tracks down. First, we perform a dictionary lookup on each of the vulnerability reports. Then, we mark dictionary words in that report as the software name, if the NER model has already identified at least one dictionary word as a software name. In this way, we can rectify some labels incorrectly identified. For example, in Figure 2, assume the NER model assigns the `SN` labels to words `Microsoft VBScript` and `Explorer` indicating the

[2]Vulnerable software names may include the names of vulnerable libraries or the affected operating systems. The names of vulnerable libraries and the affected OSes are also extracted by our tool.

software pertaining to the vulnerability. Through dictionary lookup, we track down software `Internet Explorer` in the gazetteer. Since the gazetteer indicates neither `Internet` nor `Explorer` has ever occurred individually as a software name, and they are the closest to the dictionary word `Internet Explorer`, we extend our label and mark the entire dictionary word `Internet Explorer` with an `SN` label and treat it as single software.

3.3 Relation Extraction Model

The relation extraction model was originally used to extract the relationships of two entities [41]. Over the past decade, researchers proposed various technical approaches to build highly accurate and computationally efficient RE model. Of all the techniques proposed, hierarchical attention neural networks [57] demonstrate better performance in many natural language processing tasks. For our system, we modify an existing hierarchical attention neural network to *pair* the extracted entities (*i.e.*, pairing vulnerable software names and their versions). More specifically, we implement a new *combined* word-level and sentence-level attention network in the RE model to improve the performance. In the following, we briefly introduce this model and discuss how we apply it to our problem. For more details about the RE model in general, readers could refer to these research papers [38,57,62].

As is depicted in Figure 2, the RE model pinpoints the right relationship between software name and version through a three-step procedure. In the first step, it encodes the occurrence of the software names as well as that of the version information, and then yields a group of position embeddings representing the relative distances from current word to the two named entities (*i.e.,* software name and version) in the same sentence [38]. To be specific, the RE model first indexes the sequence of the entity labels generated by the aforementioned NER model. Then, it runs through all the software names and versions in every possible combination, and encodes the combinations based on the indexes of the entity labels using one-hot encoding. With the completion of one-hot encoding, the RE model further employs a word embedding approach to convert the one-hot encoding into two individual vectors indicating the embeddings of the positions (see Figure 2).

In the second step, similar to the NER model, the RE model uses the same technique to encode text and transfers a text sequence into a sequence of vectors. Then, right behind the word sequence vectors, the RE model appends each group of the position embeddings individually. For example, in Figure 2, the NER model pinpoints two software names and two versions which form four distinct combinations (all the possible ways of pairing software names and versions). For each combination, the RE model appends the position embedding vector to the word embedding vector to form the input for the last step for performing classifications. In this example, four vectors are produced as the input, and each represents a possible software name-version pair.

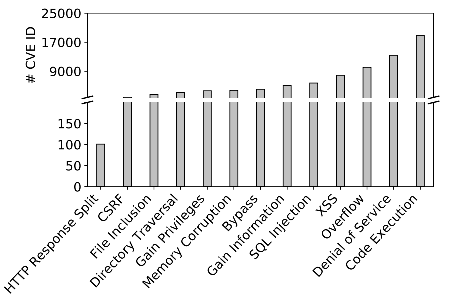


Figure 4: # of CVE IDs per vulnerability category.

In the last step, the RE model takes each sequence of vectors as the input for an attention-based neural network and outputs a vector indicating the new representation of the sequence. Then, as is illustrated in Figure 2, the RE model takes the output vectors as the input for another attention network, through which the RE model predicts which name-version pairing is most likely to capture the relationship between software name and its corresponding versions. Continue the example in Figure 2. The `seq 1` and `4` are associated with a positive output, which indicates the legitimate pairing relationships of `Microsoft VBScript 5.7 and 5.8` and `Internet Explorer 9 through 11`.

4 Dataset

To evaluate our system and detect real-world inconsistencies, we collected a large number of public vulnerability reports and CVE and NVD entries from the past 20 years. We sample a subset of these vulnerability reports for manual labeling (ground-truth) and use the labeled data to evaluate the performance of `VIEM`.

CVE IDs. We first obtain a list of CVE IDs from `cvedetails.com`, which divides the security vulnerabilities indexed in CVE/NVD database into 13 categories. To collect a representative dataset of publicly-reported vulnerabilities, we crawled the CVE IDs from January 1999 to March 2018 (over 20 years) of each vulnerability category. A CVE ID is the unique identifier for a publicly disclosed vulnerability. Even though the CVE website claimed that they have over 105,000 CVE IDs [6], many of the CVE IDs are either not publicly available yet, or have been merged or withdrawn. In total, we obtain 78,296 CVE IDs covering all 13 categories as shown in Figure 4. Each CVE ID corresponds to a short *summary* of the vulnerability as shown in Table 1.

Vulnerability Reports. The webpage of each CVE ID also contains a list of external references pointing to external reports. Our study focuses on 5 representative source websites to obtain the vulnerability reports referenced by the CVE, including ExploitDB [7], SecurityFocus [14], SecurityTracker [16], Openwall [13], and SecurityFocus Fo-

Dataset	CVE IDs	CVE Summaries (Unstructured)	NVD Entries (Structured)	Vulnerability Reports	Structured Reports		Unstructured Reports		
					SecTracker	SecFocus	ExploitDB	Openwall	SecF Forum
All	78,296	78,296	78,296	70,569	7,320	38,492	9,239	5,324	10,194
G-truth	5,193	5,193	0	1,974	0	0	785	520	669

Table 1: Dataset statistics.

rum [15]. Note that we treat SecurityFocus and SecurityFocus Forum as two different websites. SecurityFocus site only displays the "structured" information (*e.g.*, affected OS, software versions). SecurityFocus Forum (also called Bugtraq Mailing List) mainly contains "unstructured" articles and discussion threads between vulnerability reporters and software developers. Regarding the other three websites, SecurityTracker also contains well-structured information, while Openwall and ExploitDB[3] contain unstructured information. In total, we obtain 70,569 vulnerability reports associated with 56,642 CVE IDs. These CVE IDs cover 72.34% of all 78,296 public CVE IDs. This means 72.34% of the CVE IDs have a vulnerability report from one of the 5 source websites, confirming their popularity. There are 45,812 structured reports from SecurityTracker and SecurityFocus, and 24,757 unstructured reports from ExploitDB, Openwall, and SecurityFocus Forum.

NVD Entries. For each CVE ID, we also parse the JSON version of the NVD entries, which contains the structured data fields such as vulnerable software names and their versions. We obtain 78,296 NVD entries in total.

Data Extraction and Preprocessing. For *structured* reports, we directly parse the vulnerable software name and version information following the fixed format. For *unstructured* vulnerability reports and CVE summaries, we extract the text information, remove all the web links, and tokenize the sentences using the NLTK toolkit [24]. Note that we did not remove any stop words or symbols from the unstructured text, considering that they are often parts of the software names and versions.

Ground-Truth Dataset. To evaluate VIEM, we construct a "ground-truth" dataset by manually annotating the vulnerable software names and versions. As shown in Table 1, the ground-truth dataset contains only unstructured reports, covering 5,193 CVE IDs (the short summaries) and 1,974 unstructured reports. Some reports are referenced by multiple CVE IDs. We choose to label our own ground-truth dataset instead of using the structured data as the ground-truth, for two reasons. First, the structured data is not necessarily correct. Second, the NER model needs labels at the sentence level and the word level. The labels for the RE model represent the relationship between the extracted entities. The structured data cannot provide such labels.

[3]ExploitDB has some structured information such as the affected OS, but the vulnerable software version often appears in the titles of the posts and the code comments.

The 5,193 CVE IDs are not evenly sampled from different vulnerability categories. Instead, we sampled a large number of CVE IDs from one primary category and a smaller number of CVE IDs from the other 12 categories to evaluate model transferability. We choose memory corruption as the primary category for its severity and real-world impact (*e.g.*, Heartbleed, WannaCry). An analysis of the severity scores (CVSS) also shows that memory corruption vulnerabilities have the highest average severity score (8.46) among all the 13 categories. We intend to build a tool that at least performs well on memory corruption cases. Considering the high costs of manual annotation, we decide to label a large amount of data (3,448 CVE IDs) in one category (memory corruption), and only label a small amount of data (145 CVE IDs) for each of the other 12 categories. With this dataset, we can still apply transfer learning to achieve a good training result.

Given a document, we perform annotation in two steps. First, we manually label the *vulnerable software names* and *vulnerable software versions*. This step produces a ground-truth dataset to evaluate our NER model. Second, we manually *pair* the vulnerable software names with the versions. This step produces a ground-truth mapping to evaluate the RE model. We invited 6 lab-mates to perform the annotation. All the annotators have a bachelor or higher degree in Computer Science and an in-depth knowledge of Software Engineering and Security. Figure 2 shows an example. For the NER dataset, we label each word in a sentence by assigning one of the three labels: vulnerable software name (*SN*), vulnerable software version (*SV*)[4], or others (*O*). Note that entities that are related to non-vulnerable software will be labeled as *O*. For the RE dataset, we pair *SN* and *SV* entities by examining all the possible pairs *within the same sentence*. Through our manual annotation, we never observe a vulnerable software name and its versions located in completely different sentences throughout the entire document.

5 Evaluation

In this section, we use the ground-truth dataset to evaluate the performance of VIEM. First, we use the dataset of memory corruption vulnerabilities to assess the system performance and fine-tune the parameters. Then, we use the data of the other 12 categories to examine the model transferability.

[4]For software versions, we treat keywords related to compatibility pack, service pack, and release candidate (*e.g.*, business, express edition) as part of the version. For versions that are described as a "range", we include the conjunctions in the version labels.

Metric		w/o Gazetteer	w/ Gazetteer
Software Version	Precision	0.9880	0.9880
	Recall	0.9923	0.9923
Software Name	Precision	0.9773	0.9782
	Recall	0.9916	0.9941
Overall	Accuracy	0.9969	0.9970

Table 2: NER performance on "memory corruption" dataset.

5.1 Evaluating the NER and RE Model

To evaluate the NER and RE models, we use the *memory corruption* vulnerability reports and their CVE summaries (3,448 CVE IDs).

NER Model. Given a document, the NER model extracts the vulnerable software names and vulnerable versions. The extraction process is first at the *word level*, and then the consecutive words with the *SN* or *SV* label will be grouped into software names or software versions. We use three evaluation metrics on the word-level extraction: (1) *Precision* represents the fraction of the relevant entities over the extracted entities; (2) *Recall* represents the fraction of the relevant entities that are extracted over the total number of relevant entities; (3) *Overall accuracy* represents the fraction of the correct predictions over all the predictions. We compute the precision and recall for software name extraction and version extraction separately.

We split the ground-truth dataset with a ratio of 8:1:1 for training, validation, and testing. We use a set of default parameters, and later we show that our performance is not sensitive to the parameters. Here, the dimension of the pre-trained word embeddings is 300 (50 for character embeddings). To align our input sequences, we only consider the first 200 words per sentence. Empirically, we observe that the vast majority of sentences are shorter than 200 words. All the layers in the NER model are trained jointly except the word-level embedding weight W (using the `FastText` method). The default batch size is 50 and the number of epochs is 20. We use an advanced stochastic gradient descent approach `Adam` as the optimizer, which can adaptively adjust the learning rate to reduce the convergence time. We also adopt `dropout` to prevent overfitting.

We repeat the experiments 10 times by randomly splitting the dataset for training, validation, and testing. We show the average precision, recall, and accuracy in Table 2. Our NER model is highly accurate even without applying the gazetteer (*i.e.*, the dictionary). Both vulnerable software names and versions can be extracted with a precision of 0.978 and a recall of 0.991. In addition, we show that gazetteer can improve the performance of software name extraction as expected. After applying the gazetteer, the overall accuracy is as high as 0.9969. This high accuracy of NER is desirable because any errors could propagate to the later RE model.

We also observe that our NER model indeed extracts software names (and versions) that *never appear* in the training dataset. For example, after applying NER to the testing dataset, we extracted 205 unique software names, and 47 (22.9%) of them never appear in the training dataset. This confirms that the NER model has learned generalizable patterns and indicators to extract relevant entities, which allows the model to extract *previously unseen* software names.

Metric	Ground-truth Software Name/Version as Input	NER Model's Result as Input	
		w/o Gazetteer	w/ Gazetteer
Precision	0.9955	0.9248	0.9411
Recall	0.9825	0.9931	0.9932
Accuracy	0.9916	0.9704	0.9764

Table 3: RE performance on "memory corruption" dataset.

RE Model. We then run experiments to first examine the performance of the RE model itself, and then evaluate the end-to-end performance by combining NER and RE. Similar as before, we split the ground-truth dataset with an 8:1:1 ratio for training, validation, and testing. Here, we set the dimension of the pre-trained word embeddings to 50. The dimension of position embeddings is 10. The default batch size is 80 and the number of epochs is 200. We set the number of bi-directional layers as 2. Like the NER model, our RE model also uses the pre-trained word embedding weight W. The position embedding weights (*i.e.,* W_s and W_v) are randomly initialized and trained together with other parameters in the model.

First, we perform an experiment to evaluate the RE model alone. More specifically, we assume the named entities are already correctly extracted, and we only test the "pairing" process using RE. This assumes the early NER model has a perfect performance. As shown in Table 3, the RE model is also highly accurate. The model has a precision of 0.9955 and a recall of 0.9825.

Second, we evaluate the end-to-end performance, and use the NER's output as the input of the RE model. In this way, NER's errors may affect the performance of RE. As shown in Table 3, the accuracy has decreased from 0.9916 to 0.9704 (without gazetteer) and 0.9764 (with gazetteer). The degradation mainly happens in precision. Further inspection shows that the NER model has falsely extracted a few entities that are not software names, which then become the false input for RE and hurt the classification precision. In addition, the benefit of gazetteer also shows up after NER and RE are combined, bumping the precision from 0.9248 to 0.9411 (without hurting the recall). The results confirm that our model is capable of accurately extracting vulnerable software names and the corresponding versions from unstructured text.

Baseline Comparisons. To compare the performance of `VIEM` against other baselines, we apply other methods to the same dataset. First, for the NER model, we tested SemFuzz [59]. SemFuzz uses hand-built regular expressions to extract vulnerable software versions from vulnerability re-

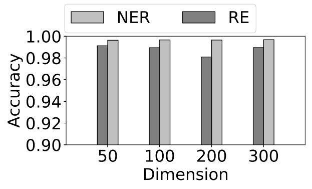


Figure 5: Word embedding dimension vs. accuracy.

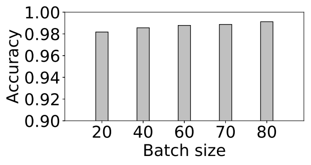


Figure 6: Batch size vs. RE model accuracy.

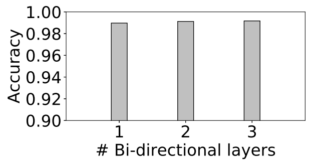


Figure 7: # network layers vs. RE model accuracy.

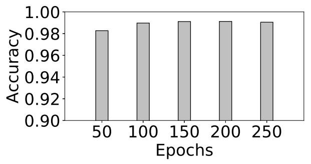


Figure 8: # Epochs vs. RE model accuracy.

ports. We find that SemFuzz achieves a reasonable precision (0.8225) but a very low recall (0.371). As a comparison, our precision and recall are both above 0.978. Second, for the end-to-end evaluation, we implemented a baseline system using off-the-shelf NLP toolkits. More specifically, we use the Conditional Random Field sequence model for extracting named entities. The model uses Stanford Part-Of-Speech tags [52] and other syntactic features. Then we feed the extracted entities to a baseline RE model that is trained with features from Stanford Neural Dependency Parsing [29]. The end-to-end evaluation returns a precision of 0.8436 and a recall of 0.8851. The results confirm the better performance of `VIEM`, at least for our application purposes (*i.e.*, processing vulnerability reports).

Model Parameters. The above results are based on a set of default parameters that have been fine-tuned based on the validation dataset. To justify our parameter choices, we change one parameter at a time and see its impact on the model. We perform this test on all parameters (*e.g.,* word embedding dimensions, batch sizes, network layers, epochs). The takeaway is that our model is not very sensitive to the parameter settings.

We pre-trained our own word embedding layer on the corpus built from all the unstructured reports in our dataset. We have tested two state-of-the-art methods `Word2vec` [39], and `FastText` [25]. We choose `FastText` since it gives a slightly higher accuracy (by 1%). Figure 5 shows the overall accuracy of `FastText` embeddings under different embedding dimensions. The results show that the performance is not sensitive to this parameter (as long as it is configured within a reasonable range). When training RE and NER, we need to set the batch size, the number of epochs, the number of bi-directional layers and the dimension of position embeddings in the neural networks. Again, we swap the parameters for RE and NER separately. For brevity, we only show the results for the RE model in Figure 6–Figure 8. The results for the NER model are similar, which are shown in Appendix-A.

Metric		Before Transfer		After Transfer	
		w/o Gaze	w/ Gaze	w/o Gaze	w/ Gaze
Software Version	Precision	0.8428	0.8467	0.9382	0.9414
	Recall	0.9407	0.9400	0.9410	0.9403
Software Name	Precision	0.8278	0.8925	0.9184	0.9557
	Recall	0.8489	0.9185	0.9286	0.9536
Overall	Accuracy	0.9873	0.9899	0.9942	0.9952

Table 4: Transfer learning result of NER model (average over 12 vulnerability categories).

5.2 Evaluating Transfer Learning

Finally, we examine the generalizability of the model to other vulnerability categories. First, to establish a baseline, we *directly* apply the model trained with memory corruption dataset to other vulnerability categories. Second, we apply transfer learning, and retrain a dedicated model for each of the other vulnerability categories. We refer the two experiments as "before transfer" and "after transfer" respectively.

Transfer learning is helpful when there is not enough labeled data for each of the vulnerability categories. In this case, we can use the memory corruption classifier as the teacher model. By fine-tuning *the last layer* of the teacher model with the data of each category, we can train a series of category-specific classifiers. More specifically, we train a teacher model using all the ground-truth data from the memory corruption category (3,448 CVE IDs). Then we use the teacher model to train a new model for each of the 12 vulnerability categories. Given a target category (*e.g.*, SQL Injection), we split its ground-truth data with a ratio of 1:1 for training (T_{train}) and testing (T_{test}). The training data (T_{train}) will be applied to the pre-trained teacher model to fine tune the final hidden layer. In this way, a new model that is specially tuned for "SQL Injection" reports is constructed. Finally, we apply this new model to the testing data (T_{test}) to evaluate its performance.

The results are presented in Table 4 and Table 5. We find that the NER model is already highly generalizable before transfer learning. Transfer learning only introduces a small improvement in accuracy (from 0.987 to 0.994). Second, the RE model (trained on memory corruption data) has a clear degradation in performance when it is directly applied to other categories. The overall accuracy is only 0.876. After transfer learning, accuracy can be improved to 0.9044.

To confirm that transfer learning is necessary, we run an additional experiment by using all the ground-truth dataset from 13 categories to train a single model. We find that the end-to-end accuracy is only 0.883 which is lower than the transfer learning accuracy. The accuracy of certain categories clearly drops (*e.g.*, 0.789 for CSRF). This shows that the vulnerability reports of different categories indeed have different characteristics, and deserve their own models.

Metric	Before Transfer			After Transfer		
	G-truth	NER Result as Input		G-truth	NER Result as Input	
	as Input	w/o Gaze	w/ Gaze	as Input	w/o Gaze	w/ Gaze
Precis.	0.9559	0.7129	0.8105	0.9781	0.8062	0.8584
Recall	0.9521	0.9767	0.9724	0.9937	0.9964	0.9964
Accur.	0.9516	0.8215	0.8760	0.9834	0.8698	0.9044

Table 5: Transfer learning result of RE model (average over 12 vulnerability categories.)

6 Measuring Information Inconsistency

In this section, we apply VIEM to the full dataset to examine the information consistency. In particular, we seek to examine how well the structured NVD entries are matched up with the CVE entries and the referenced vulnerability reports. In the following, we first define the metrics to quantify consistency. Then we perform a preliminary measurement on the ground-truth dataset to estimate the measurement errors introduced by VIEM. Finally, we apply our model to the full dataset to examine how the consistency level differs across different vulnerability types and over time. In the next section (§7), we will perform case studies on the detected inconsistent reports, and examine the causes of the inconsistency.

6.1 Measurement Methodology

NVD database, with its fully *structured* and *standardized* data entries, makes it possible for automated information processing and intelligent mining. However, given that NVD entries are created and updated by manual efforts [18], we are concerned about its data quality, in particular, its ability to keep up with the most recent discoveries of vulnerable software and versions. To this end, we seek to measure the consistency of NVD entries with other information sources (including CVE summaries and external reports).

Matching Software Names. Given a CVE ID, we first match the vulnerable software names listed in the NVD database, and those from unstructured text. More specifically, let $C = \{(N_1,V_1),(N_2,V_2),...,(N_n,V_n)\}$ be the vulnerable software name-version tuples extracted from the NVD, and $C' = \{(N_1,V_1'),(N_2,V_2'),...,(N_m,V_m')\}$ be the name-version tuples extracted from the external text. In our dataset, about 20% of the CVE IDs are associated with multiple software names. In this paper, we only focus on the *matched software names* between the NVD and external reports. Our matching method has the flexibility to handle the slightly different format of the same software name. We consider two names as a match if the number of matched words is higher or equal to the number of unmatched words. For example, "Microsoft Internet Explorer" and "Internet Explorer" are matched because there are more matched words than the unmatched one.

Measuring Version Consistency. Given a software name N_1, we seek to measure the consistency of the reported versions V_1 and V_1'. We examine two types of matching. First, *strict matching* means V_1 and V_1' exactly match each other ($V_1 = V_1'$). Second, *loose matching* means one version is the other version's superset ($V_1 \subseteq V_1'$ or $V_1 \supseteq V_1'$). Note that the loosely matched cases contain those that are strictly matched. Beyond loose matching, it means V_1 and V_1' each contains some vulnerable versions that are not reported by the other (*i.e.*, conflicting information).

To perform the above matching procedure, we need to convert the text format of V_1 and V_1' to a comparable format. In unstructured text, the software version is either described as a set of discrete values (*e.g.*, "version 1.1 and 1.4") or a continuous range (*e.g.*, "version 1.4 and earlier"). For descriptions like "version 1.4 and earlier", we first convert the text representation into a mathematical range. The conversion is based on a dictionary that we prepared beforehand. For example, we convert "... and earlier" into "$\leq$". This means "1.4 and earlier" will be converted into "$\leq$ 1.4". Then, for "range" based version descriptions, we look up the CPE directory maintained by NIST [2] to obtain a list of all the available versions for a given software. This allows us to convert the "range" description ("$\leq$ 1.4") into a set of discrete values $\{1.0, 1.1, 1.2, 1.3, 1.4\}$. After the conversion, we can determine if V_1 and V_1' match or not.

If a CVE ID has more than one software names ($k > 1$), we take a conservative approach to calculate the matching result. Only if all the k software version pairs are qualified as strict matching will we consider the report as a "strict match". Similarly, only if all the pairs are qualified as loose matching will we label the report as "loose matching" report.

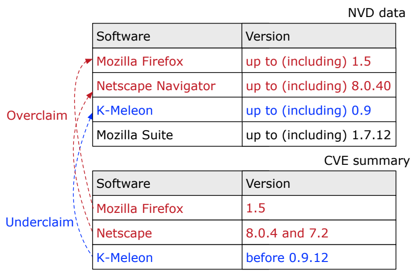
Match	Memory Corruption			Avg. over 12 Other Categories		
	Matching Rate		Deviat.	Matching rate		Deviat.
	VIEM	G-truth		VIEM	G-truth	
Loose	0.8725	0.8528	0.0194	0.9325	0.9371	-0.0046
Strict	0.4585	0.4627	-0.0042	0.6100	0.6195	-0.0095

Table 6: Strict matching and loose matching results on the ground-truth dataset.

6.2 Ground-truth Measurement

Following the above methodology, we first use the ground-truth dataset to estimate the measurement error. More specifically, given all the ground-truth vulnerability reports and the CVE summaries, we use our best-performing model (with gazetteer and transfer learning) to extract the vulnerable software name-version tuples. Then we perform the strict and loose matching on the extracted entries and compare the matching rate with the ground-truth matching rate. The results are shown in Table 6.

We show that VIEM introduced a small deviation to the actual matching rate. For memory corruption vulnerabilities, our model indicates that 87.3% of the reports loosely match the NVD entries, and only 45.9% of the reports strictly



NVD data

Software	Version
Mozilla Firefox	up to (including) 1.5
Netscape Navigator	up to (including) 8.0.40
K-Meleon	up to (including) 0.9
Mozilla Suite	up to (including) 1.7.12

CVE summary

Software	Version
Mozilla Firefox	1.5
Netscape	8.0.4 and 7.2
K-Meleon	before 0.9.12

Figure 9: Example of underclaimed and overclaimed versions.

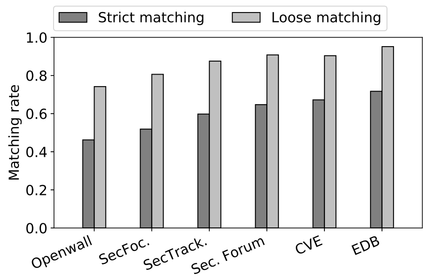


Figure 10: Matching rate for different information sources.

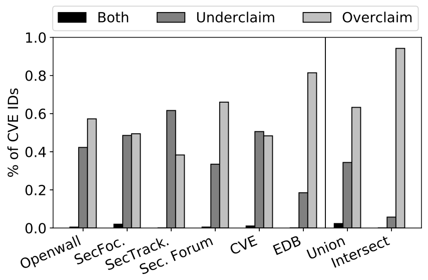


Figure 11: Breakdown of underclaimed and overclaimed cases.

match. The results are very close to the ground-truth where the matching rates are 85.3% and 46.3% respectively. For the rest of the 12 vulnerability categories, our model indicates that the loose matching rate is 93.3% and the strict matching rate is 61%. Again, the results are quite similar to the ground-truth (93.7% and 62%). The deviation from ground-truth ($Rate_{estimated} - Rate_{groundtruth}$) is bounded within $\pm1.9\%$. The results confirm that our model is accurate enough for the measurement.

6.3 Large-Scale Empirical Measurements

After the validation above, we then use the full ground-truth dataset to train `VIEM` and apply the model to the rest of the unlabeled and unstructured text (vulnerability reports and CVE summaries). Then we calculate the matching rate between the versions of NVD and those from external information sources (the CVE website and 5 external websites).

Result Overview. Across all 78,296 CVE IDs, we extract in total 18,764 unique vulnerable software names. These vulnerable software names correspond to 154,569 software name-version pairs from the CVE summaries, 235,350 name-version pairs from the external vulnerability reports, and 165,822 name-version pairs from NVD database. After matching the software names between NVD and other sources, there are 389,476 pairs left to check consistency.

At the name-version pair level, we find 305,037 strictly matching pairs (78.32%). This means about 22% of the name-version pairs from NVD do not match the external information sources. If we relax the matching condition, we find 361,005 loosely matched pairs (93.49%).

We then aggregate the matching results at the *report level*. Although the loose matching rate is still high (90.05%), the strict matching rate clearly decreases. Only 59.82% of the vulnerability reports/CVE summaries strictly match the NVD entries. This is because strictly matched reports require all the extracted versions to match those of NVD.

In order to understand how the level of consistency varies across different aspects, we next break down the results for more in-depth analyses.

Information Source Websites. Figure 10 shows the matching rates between the NVD entries and the 5 information websites and the CVE website. CVE has a relatively high matching rate (about 70% strict matching rate). This is not too surprising given that NVD is claimed to be synchronized with the CVE feed. More interestingly, we find that ExploitDB has an even higher matching rate with NVD. We further examine the *posting dates* of the NVD entries and the corresponding reports in other websites. We find that the vast majority (95.8%) of the ExploitDB reports were posted after the NVD entries were created. However, 81% of the ExploitDB reports were posted earlier than the reports in the other 4 websites, which might have helped to catch the attention of the NVD team to make an update.

Overclaims vs. Underclaims. For the loosely matched versions, the NVD entries may have overclaimed or underclaimed the vulnerable software versions with respect to the external reports. An example is shown in Figure 9 for `CVE-2005-4134`. Compared with CVE summary, NVD has *overclaimed* the vulnerable version for *Mozilla Firefox* and *Netscape Navigator*, given that NVD listed more vulnerable versions than CVE. On the contrary, for *K-Meleon*, NVD has *underclaimed* the vulnerable software version range.

Figure 11 shows the percentage of overclaimed and underclaimed NVD entries within the loosely matched pairs. "Strict-matched" pairs are not included in this analysis. We are not surprised to find NVD entries might overclaim. Given that NVD is supposed to search for different sources to keep the entries update-to-date, it is reasonable for the NVD entries to cover more vulnerable versions. Even if we take the union of the vulnerable versions across 5 websites and CVE, NVD is still covering more versions. The more interesting observation is that NVD has underclaimed entries compared to each of the external information sources. This suggests that NVD is either suffering from delays to update the entries or fails to keep track of the external information. Only a tiny portion of NVD entries contain both overclaimed and underclaimed versions (see the example in Figure 9).

Examples of Conflicting Cases. We then examined the conflicted pairs, and observed that a common reason for mis-

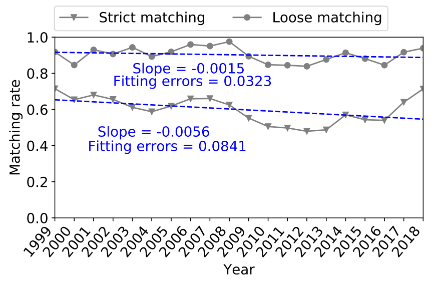


Figure 12: Matching rate over time: NVD vs. (CVE + 5 websites).

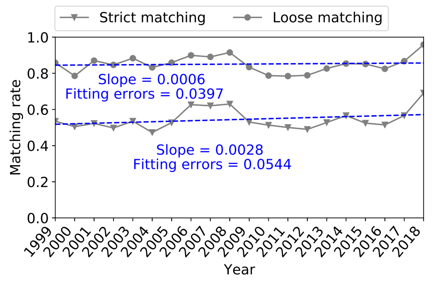


Figure 13: Matching rate over time: CVE vs. 5 websites.

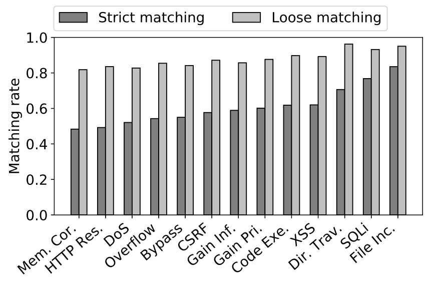


Figure 14: Matching rate for different vulnerability categories.

matching is *typos*. For example, under CVE-2008-1862, the versions listed on CVE and ExploitDB are both `0.22 and earlier` for software *ExBB Italia*. However, the NVD version is slightly different "`up to (including) 0.2.2`". Another example is CVE-2010-0364 where the software versions from NVD and CVE summary are both `0.8.6` for *Videolan VLC media player*. However, the information on SecurityFocus has a clear typo as `0.6.8`.

In other cases, it is not clear whose information is correct. For example, sometimes the referenced vulnerable reports provide more detailed information than the CVE summary. For example, under CVE-2012-1173, the vulnerable version for *libtiff* is listed as `3.9.4` on NVD but SecurityTracker claims the vulnerable version should be `3.9.5`. Under CVE-2000-0773, software *Bajie HTTP web server* is vulnerable for version `1.0` according to NVD. However, CVE lists version `0.30a` and SecurityFocus lists `0.90, 0.92, 0.93`. There is no way to determine the correctness of the contradicting information at the pure text level, but we argue that the value of such measurement results is to point out the cases that need validation and correction.

Consistency Over Time. Figure 12 shows the consistency level between NVD and the other 6 information sources (CVE and the 5 report websites) is decreasing over time. The strict matching rate has some fluctuation over time but still shows a decreasing trend. We perform a linear regression for both matching rates and find both have a negative slope (-0.0015 and -0.0056 respectively). The result suggests the overall consistency drops over time in the past 20 years. However, if we take a closer look at the recent three years (2016 to 2018), the consistency level is starting to increase, which is a good sign.

Figure 13 shows a different trend when we compare the consistency between CVE and the 5 external websites. The consistency level between CVE and external sites is relatively stable with a slight upward trend. We perform a linear regression for both matching rates which returns a positive slope (0.0006 and 0.0028 respectively). This suggests that CVE websites are getting better at summarizing the vulnerability versions.

Types of Vulnerabilities. As shown in Figure 14, we break down the results based on the vulnerability categories. While the loose matching rates are still similar (around 90%), there are clear differences in their strict matching rates. For example, "SQL Injection" and "File Inclusion" have the highest strict matching rate (over 75%), but categories such as "Memory Corruption" have a much lower strict matching rate (48%). Further manual examination suggests that memory corruption vulnerabilities are typically more complex than those under File Inclusion or SQL Injection, and thus require a longer time to reproduce and validate. As a result, it is not uncommon for NVD to miss newly discovered vulnerable versions over time.

Inferring the Causes of Inconsistencies. Finally, we attempt to infer the causes of inconsistencies by analyzing the NVD entry creation/update time with respect to the posting time of the external reports. More specifically, NVD maintains a "change history" for each CVE ID, which allows us to extract the entry creation time, and the time when new software versions are added/removed. Then we can compare it with the posting time of corresponding reports at the 5 websites. For this analysis, we randomly select 5,000 CVE IDs whose vulnerable versions in NVD are inconsistent with those of the 5 websites.

We find that 66.3% of the NVD entries have never been updated since they were created for the first time. This includes 5.8% NVD entries that were created before any of the 5 websites posted their reports. For example, for CVE-2006-6516, NVD claimed *KDPics* `1.16` was vulnerable in 2006. Later in 2010, SecurityFocus reported that both version `1.11` and version `1.16` were vulnerable. NVD has not added the new version `1.11` until today. For the much bigger portion (60.5%) of NVD entries, they were created when at least one of the external reports were already available. An example is CVE-2016-6855 as ExploitDB claimed *Eye of Gnome* `3.10.2` was vulnerable in August 2016. A month later, the NVD entry was created which did not include version `3.10.2`. No update has been made since then.

For the rest of the 33.7% of the NVD entries, they have made at least one update to the vulnerable versions after the entry creation. For them, we compare the latest update time

CVE ID	Software	Vul. Versions Claimed by Different Sources	Majority Vote	Union	Ground truth	Versions Manually Tested	# Newly Detected Versions by Us (12)	# Overclaimed Reports (15)
CVE-2004-2167	latex2rtf	**NVD:** 1.9.15 (1) **CVE:** 1.9.15 and possibly others (40) **SecurityFocus, SecurityTracker:** 1.9.15 (1) **IBM Security:** ≤ 1.9.15 (14)	1.9.15 (1)	1.9.15 and possibly others (40)	1.9.15 (1)	1.8aa - 2.3.17 (40)	0	1
CVE-2008-2950	poppler	**NVD, CVE, SecurityTracker, Security Forum, OCERT, CXsecurity, IBM Security:** ≤ 0.8.4 (34) **SecurityFocus, ExploitDB:** 0.8.4 (1) **RedHat:** < 0.6.2 (22) **Gentoo:** < 0.6.3 (23)	≤ 0.8.4 (34)	≤ 0.8.4 (34)	0.5.9 - 0.8.4 (16)	0.1 - 0.8.7 (37)	0	7
CVE-2009-5018	gif2png	**NVD:** 0.99 - 2.5.3 (36) **CVE, Openwall, IBM Security, Bugzilla:** ≤ 2.5.3 (36) **SecurityFocus:** 2.5.2 (1) **Gentoo:** < 2.5.1 (33) **Fedora:** 2.5.1 (1)	≤ 2.5.3 (36)	≤ 2.5.3 (36)	2.4.2 - 2.5.6 (13)	0.7 - 2.5.8 (41)	2.5.4 - 2.5.6 (3)	4
CVE-2015-7805	libsndfile	**NVD, CVE, Openwall, Fedora, nemux, Packet Storm:** 1.0.25 (1) **ExploitDB:** ≤ 1.0.25 (30) **Gentoo:** < 1.0.26 (30)	1.0.25 (1)	≤ 1.0.25 (30)	1.0.15 - 1.0.25 (11)	0.0.8 - 1.0.26 (31)	0	2
CVE-2016-7445	openjpeg	**NVD, Gentoo:** ≤ 2.1.1 (16) **SecurityFocus, Openwall:** 2.1.1 (1) **CVE:** < 2.1.2 (16)	2.1.1 (1)	< 2.1.2 (16)	1.5 - 2.1.1 (7)	≤ 2.2.0 (18)	0	1
CVE-2016-8676	libav	**NVD:** ≤ 11.8 (47) **CVE:** 11.9 (1) **SecurityFocus:** 11.3, 11.4, 11.5, 11.7 (4) **Openwall:** 11.8 (1) **agostino's blog**: 11.3 - 11.7 (5)	11.3, 11.4, 11.5, 11.7 (4)	11.3, 11.4, 11.5, 11.7, 11.8, 11.9 (6)	11.0 - 11.8 (9)	11.0 - 11.9 (10)	11.0, 11.1, 11.2, 11.6 (4)	0
CVE-2016-9556	ImageMagick	**NVD, CVE:** 7.0.3.8 (1) **SecurityFocus, Openwall, agostino's blog:** 7.0.3.6 (1)	7.0.3.6	7.0.3.6, 7.0.3.8 (2)	7.0.3.1 - 7.0.3.7 (7)	7.0.3.1 - 7.0.3.8 (8)	7.0.3.1 - 7.0.3.5 (5)	0

Table 7: The summary of case study results. The number in parentheses denotes the total number of software versions.

and the posting time of the external reports at the 5 websites. We find all of the NVD entries made the latest update *after* the posting time of some of the external reports. Overall, these results suggest that the NVD team did not effectively include the vulnerable versions from the external reports, despite that the reports were already available at the time of the entry creation/update. The results in turn reflect the need of automatically monitoring different online sources and extracting vulnerable versions for more proactive version testing and entry updating.

7 Case Study

To demonstrate the real-world implications of our inconsistency measurement, we perform case studies. We randomly select 7 real-world vulnerabilities from the *mismatched cases* in our dataset. Then we attempt to manually reproduce the vulnerabilities of the related software under different versions. These vulnerabilities are associated with 7 distinct CVE IDs, covering 47 vulnerability reports in total. Note that for the case study, we not only included CVE summaries and the reports from the 5 websites, but considered all other source websites in the reference lists of these CVE IDs. For the software mentioned in these reports, we exhaustively gathered all the versions of these software programs and obtained *185* versions of software in total. We list the number of unique versions for each software in Table 7.

With the collected software, we examine the vulnerabilities in each version. We form a team of 3 security researchers to manually analyze the source code of these software programs, and dynamically verify the reported vulnerabilities by manually crafting the PoC (proof-of-concept) input. The 185 software versions took us 4 months to fully verify.

The truly vulnerable versions are listed in the "ground-truth" column in Table 7. In total, out of the 185 software versions, we confirm that 64 versions are vulnerable. *12 of the truly vulnerable versions are discovered by us for the first time*, which have never been mentioned in existing vulnerability reports, or CVE/NVD entries.

Observation 1. Erroneous Information Confirmed. By comparing with the ground-truth vulnerable versions, we confirmed that most information sources including CVE and NVD have either missed real vulnerable versions or falsely included non-vulnerable versions. There are widespread and routine overclaims and underclaims. Given that many system administrators heavily rely on the information in vulnerability reports to assess the risk of their system and determine whether they need to upgrade their software, it is a big concern that the "underclaiming" problem could leave vulnerable software systems unpatched. The overclaims, on the other hand, could have wasted significant manual efforts from security analysts in performing risk assessments.

Observation 2. Benefits and Limits of Majority Voting. Given a vulnerability, if we take a majority voting among different information sources, we can diminish the "overclaiming" issue, which, however, amplifies the "underclaiming" issue. The result suggests that system administrators and security analysts cannot simply utilize a majority voting mechanism to determine the risk of their software systems.

Observation 3. Benefits and Limits of Union. If we take a union set of all the claimed vulnerable versions, we can see the resulting vulnerable versions are having better coverage of the truly vulnerable versions. This indicates the benefit of broadly searching different online information sources and automatically extracting newly discovered vulnerable versions. While acknowledging the benefit, we also observe that the *union* approach is not perfect. First, the union set easily introduced overclaimed versions. Across all the vulnerabilities in Table 7, we find 15 external reports (not including NVD/CVE entries) where the claimed vulnerable versions turned out to be not vulnerable based on our tests (*i.e.,* overclaimed reports). Second, the union set sometimes fails to cover truly vulnerable versions. As shown in Table 7, we confirm 12 new vulnerable versions for `CVE-2009-5018`, `CVE-2016-8676`, and `CVE-2016-9556`. These vulnerable versions are discovered for the first time by us, by exhaustively testing all the available versions of the given software. In practice, our approach (testing all versions) is not scalable given the significant manual efforts required. To fully automate the vulnerability verification process is still an open challenge. Overall, the union approach at least helps to narrow down the testing space and improve the coverage of the truly vulnerable versions.

8 Discussion

Key Insights. The most important takeaway is NVD contains highly inconsistent information from external information sources and even the CVE list. The inconsistency involves both overclaiming and underclaiming problems. The implication is that system administers or security analysts cannot simply rely on the NVD/CVE information to determine the vulnerable versions of the affected software. At the very least, browsing external vulnerability reports can help to better cover the *potentially vulnerable* versions.

Our system `VIEM` makes it possible to *automatically* keep track of the information of different sources to generate a "diff" from the NVD/CVE entries periodically. This allows employees of NIST NVD and MITRE CVE to get notified when new vulnerable versions are reported in external websites, and helps them to focus on the most inconsistent or outdated entries, which potentially accelerates vulnerability testing, entry updating, and software patching. This was not possible without `VIEM`. Although vulnerability testing and verification are still largely manual efforts (automatically verifying the correctness of the vulnerability information in reports is not yet possible, which is an open problem), our main contribution is that we enable the automation for the information collection and standardization process.

Limitations. Our study has a few limitations. First, we only focus on the 5 most popular source websites in order to make the data collection process manageable (each website needs its own crawler and content parser). We argue that the 5 websites are referenced by 72.34% of all CVE IDs, and the results are representative. Future work will seek to expand the scope of the measurements. Second, our definition of "vulnerable software" is relatively broad, including all different parts that are related to (or affected by) the vulnerability (*e.g.*, dependent libraries, OSes, components, functions). One way to improve our system is to further classify the different types of "software names" (*e.g.*, differentiating vulnerable applications and the affected OSes). Finally, the scale of our case study is still limited. The 185 software versions already cost months to manually verify, and it is difficult to increase the scale further.

9 Related Work

NLP for Security. Natural Language Processing (NLP) has been applied to address different security problems. For example, researchers apply NLP to extract malware detection features from researcher papers [63] or systematically collect cyber threat intelligence from technical blogs [27,37]. Another line of work applies text analysis to mobile apps to study permission abuse and user input sanitization [33, 35, 43, 45, 46]. Finally, NLP has been used to analyze API documentation and infer security policies [55, 61].

A more relevant line of works has employed NLP techniques to facilitate the identification and assessment of software bugs [47, 51, 54, 58]. A recent work [59] proposed a method to extract relevant information from CVE to facilitate vulnerability reproduction, with a specific focus on kernel vulnerabilities. Our work is the first to build customized NLP models to extract vulnerable software names and versions from CVE and vulnerability reports. We apply the model to systematically measure information consistency.

Security Vulnerability Reports. CVE and vulnerability reports have been studied in various contexts. Breu et al. [26] showed that the interaction between software developers and bug reporters can facilitate the remediation of software bugs. Guo et al. [34] found that vulnerabilities reported by professionals with high reputations usually get fixed quicker. Bettenburg et al. [23] discovered that additional information provided by duplicated vulnerability reports can help to resolve the problem more timely. Mu et al. [42] showed that the missing information in vulnerability reports could reduce the success rate of vulnerability reproduction. The authors of [42] only tested if one of the versions is vulnerable, and we need to test all the versions for a given software to obtain the ground-truth (§7). Zhang et al. [60] used NVD data for security risk assessment. Christey et al. [31] pointed out the biased statistics in CVE. Nappa et al. [44] found missing and extraneous vulnerable versions in NVD data (for 1500 vulnerabilities) by comparing the NVD version with those of the product security advisories. Our work differs from previous works by focusing on the information consistency between *unstructured* information sources and CVE/NVD entries for a large number of vulnerabilities.

10 Conclusion

In this paper, we design and develop an automated tool VIEM to conduct a large-scale measurement of the information consistency between the structured NVD database and unstructured CVE summaries and external vulnerability reports. Our results demonstrate that inconsistent information is highly prevalent. In-depth case studies confirm that the NVD/CVE database and third-party reports have either missed truly vulnerable software versions or falsely included non-vulnerable versions. The erroneous information could leave vulnerable software unpatched, or increase the manual efforts of security analysts for risk assessment. We believe there is an emerging need for the community to systematically rectify inaccurate claims in vulnerability reports.

Acknowledgments

We would like to thank our shepherd Giancarlo Pellegrino and the anonymous reviewers for their helpful feedback. We also want to thank Dongliang Mu and Jing Wang for their insightful suggestions. This project was supported in part by CSC scholarship, National Key Research and Development Program of China (2016YFB0800703), Chinese National Natural Science Foundation (U1836210), and NSF grants (CNS-1750101, CNS-1717028, CNS-1718459). Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of any funding agencies.

References

[1] Are there references available for cve entries? https://cve.mitre.org/about/faqs.html#cve_entry_references.

[2] CPE dictionary. https://nvd.nist.gov/products/cpe.

[3] CVE and NVD Relationship. https://cve.mitre.org/about/cve_and_nvd_relationship.html.

[4] CVE List. https://cve.mitre.org/cve/.

[5] Cve numbering authorities. https://cve.mitre.org/cve/cna.html.

[6] CVE Website. https://cve.mitre.org/.

[7] Exploitdb. https://www.exploit-db.com/.

[8] How are the cve entry descriptions created or compiled? https://cve.mitre.org/about/faqs.html#cve_entry_descriptions_created.

[9] How does a vulnerability or exposure become a cve entry? https://cve.mitre.org/about/faqs.html#cve_list_vulnerability_exposure.

[10] List of products. https://www.cvedetails.com/product-list.php.

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

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

[13] Openwall. http://www.openwall.com/.

[14] Securityfocus. https://www.securityfocus.com/vulnerabilities.

[15] Securityfocus forum. https://www.securityfocus.com/archive/1.

[16] Securitytracker. https://securitytracker.com/.

[17] Vulnerability change record for cve-2014-9662. https://nvd.nist.gov/vuln/detail/CVE-2014-9662/change-record?changeRecordedOn=6%2f30%2f2016+1%3a55%3a54+PM&type=new#0.

[18] What happens after a vulnerability is identified? https://nvd.nist.gov/general/faq#1dc13c24-565f-46eb-90da-c5cac28a1e17.

[19] What is mitre's role in cve? https://cve.mitre.org/about/faqs.html#MITRE_role_in_cve.

[20] A cyberattack the world isn't ready for. The New York Times, 2017. https://www.nytimes.com/2017/06/22/technology/ransomware-attack-nsa-cyberweapons.html.

[21] Giant equifax data breach: 143 million people could be affected. CNN, 2017. https://money.cnn.com/2017/09/07/technology/business/equifax-data-breach/index.html.

[22] ANGELI, G., PREMKUMAR, M. J. J., AND MANNING, C. D. Leveraging linguistic structure for open domain information extraction. In *Proc. of ACL* (2015).

[23] BETTENBURG, N., PREMRAJ, R., ZIMMERMANN, T., AND KIM, S. Duplicate bug reports considered harmful… really? In *Proc. of ICSM* (2008).

[24] BIRD, S., AND LOPER, E. Nltk: the natural language toolkit. In *Proc. of ACL* (2004).

[25] BOJANOWSKI, P., GRAVE, E., JOULIN, A., AND MIKOLOV, T. Enriching word vectors with subword information. *Transactions of the Association for Computational Linguistics* (2017).

[26] BREU, S., PREMRAJ, R., SILLITO, J., AND ZIMMERMANN, T. Information needs in bug reports: improving cooperation between developers and users. In *Proc. of CSCW* (2010).

[27] CATAKOGLU, O., BALDUZZI, M., AND BALZAROTTI, D. Automatic extraction of indicators of compromise for web applications. In *Proc. of WWW* (2016).

[28] CHAPARRO, O., LU, J., ZAMPETTI, F., MORENO, L., DI PENTA, M., MARCUS, A., BAVOTA, G., AND NG, V. Detecting missing information in bug descriptions. In *Proc. of FSE* (2017).

[29] CHEN, D., AND MANNING, C. A fast and accurate dependency parser using neural networks. In *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)* (2014), pp. 740–750.

[30] CHITICARIU, L., LI, Y., AND REISS, F. R. Rule-based information extraction is dead! long live rule-based information extraction systems! In *Proc. of EMNLP* (2013).

[31] CHRISTEY, S., AND MARTIN, B. Buying into the bias: Why vulnerability statistics suck. *BlackHat, Las Vegas, USA, Tech. Rep* (2013).

[32] DAVIES, S., AND ROPER, M. What's in a bug report? In *Proc. of ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ISESE)* (2014).

[33] GORLA, A., TAVECCHIA, I., GROSS, F., AND ZELLER, A. Checking app behavior against app descriptions. In *Proc. of ICSE* (2014).

[34] GUO, P. J., ZIMMERMANN, T., NAGAPPAN, N., AND MURPHY, B. Characterizing and predicting which bugs get fixed: an empirical study of microsoft windows. In *Proc. of ICSE* (2010).

[35] HUANG, J., LI, Z., XIAO, X., WU, Z., LU, K., ZHANG, X., AND JIANG, G. Supor: Precise and scalable sensitive user input detection for android apps. In *Proc. of USENIX Security* (2015).

[36] LAMPLE, G., BALLESTEROS, M., SUBRAMANIAN, S., KAWAKAMI, K., AND DYER, C. Neural architectures for named entity recognition. In *Proc. of NAACL-HLT* (2016).

[37] LIAO, X., YUAN, K., WANG, X., LI, Z., XING, L., AND BEYAH, R. Acing the ioc game: Toward automatic discovery and analysis of open-source cyber threat intelligence. In *Proc. of CCS* (2016).

[38] LIN, Y., SHEN, S., LIU, Z., LUAN, H., AND SUN, M. Neural relation extraction with selective attention over instances. In *Proc. of ACL* (2016).

[39] MIKOLOV, T., CHEN, K., CORRADO, G., AND DEAN, J. Efficient estimation of word representations in vector space. *arXiv:1301.3781* (2013).

[40] MIKOLOV, T., SUTSKEVER, I., CHEN, K., CORRADO, G. S., AND DEAN, J. Distributed representations of words and phrases and their compositionality. In *Proc. of NIPS* (2013).

[41] MINTZ, M., BILLS, S., SNOW, R., AND JURAFSKY, D. Distant supervision for relation extraction without labeled data. In *Proc. of ACL* (2009).

[42] MU, D., CUEVAS, A., YANG, L., HU, H., XING, X., MAO, B., AND WANG, G. Understanding the reproducibility of crowd-reported security vulnerabilities. In *Proc. of USENIX Security* (2018).

[43] NAN, Y., YANG, M., YANG, Z., ZHOU, S., GU, G., AND WANG, X. Uipicker: User-input privacy identification in mobile applications. In *Proc. of USENIX Security* (2015).

[44] NAPPA, A., JOHNSON, R., BILGE, L., CABALLERO, J., AND DUMITRAS, T. The attack of the clones: A study of the impact of shared code on vulnerability patching. In *Proc. of S&P* (2015).

[45] PANDITA, R., XIAO, X., YANG, W., ENCK, W., AND XIE, T. Whyper: Towards automating risk assessment of mobile applications. In *Proc. of USENIX Security* (2013).

[46] QU, Z., RASTOGI, V., ZHANG, X., CHEN, Y., ZHU, T., AND CHEN, Z. Autocog: Measuring the description-to-permission fidelity in android applications. In *Proc. of CCS* (2014).

[47] SAHA, R. K., LEASE, M., KHURSHID, S., AND PERRY, D. E. Improving bug localization using structured information retrieval. In *Proc. of ASE* (2013).

[48] SMITH, A., AND OSBORNE, M. Using gazetteers in discriminative information extraction. In *Proc. of Computational Natural Language Learning* (2006).

[49] STANOVSKY, G., MICHAEL, J., ZETTLEMOYER, L., AND DAGAN, I. Supervised open information extraction. In *Proc. of ACL* (2018).

[50] SUTSKEVER, I., VINYALS, O., AND LE, Q. V. Sequence to sequence learning with neural networks. In *Proc. of NIPS* (2014).

[51] TAN, L., YUAN, D., KRISHNA, G., AND ZHOU, Y. /* icomment: Bugs or bad comments?*. In *ACM SIGOPS Operating Systems Review* (2007).

[52] TOUTANOVA, K., KLEIN, D., MANNING, C. D., AND SINGER, Y. Feature-rich part-of-speech tagging with a cyclic dependency network. In *Proceedings of the 2003 Conference of the North American Chapter of the Association for Computational Linguistics on Human Language Technology-Volume 1* (2003), Association for Computational Linguistics, pp. 173–180.

[53] WANG, H., WANG, C., ZHAI, C., AND HAN, J. Learning online discussion structures by conditional random fields. In *Proc. of SIGIR* (2011).

[54] WONG, C.-P., XIONG, Y., ZHANG, H., HAO, D., ZHANG, L., AND MEI, H. Boosting bug-report-oriented fault localization with segmentation and stack-trace analysis. In *Proc. of International Conference on Software Maintenance and Evolution (ICSME)* (2014).

[55] XIAO, X., PARADKAR, A., THUMMALAPENTA, S., AND XIE, T. Automated extraction of security policies from natural-language software documents. In *Proc. of SIGSOFT* (2012).

[56] YANG, Z., SALAKHUTDINOV, R., AND COHEN, W. W. Transfer learning for sequence tagging with hierarchical recurrent networks. *arXiv preprint arXiv:1703.06345* (2017).

[57] YANG, Z., YANG, D., DYER, C., HE, X., SMOLA, A., AND HOVY, E. Hierarchical attention networks for document classification. In *Proc. of NAACL-HLT* (2016).

[58] YE, X., BUNESCU, R., AND LIU, C. Learning to rank relevant files for bug reports using domain knowledge. In *Proc. of SIGSOFT* (2014).

[59] YOU, W., ZONG, P., CHEN, K., WANG, X., LIAO, X., BIAN, P., AND LIANG, B. Semfuzz: Semantics-based automatic generation of proof-of-concept exploits. In *Proc. of CCS* (2017).

[60] ZHANG, S., OU, X., AND CARAGEA, D. Predicting cyber risks through national vulnerability database. *Information Security Journal: A Global Perspective* (2015).

[61] ZHONG, H., ZHANG, L., XIE, T., AND MEI, H. Inferring resource specifications from natural language api documentation. In *Proc. of ASE* (2009).

[62] ZHOU, P., SHI, W., TIAN, J., QI, Z., LI, B., HAO, H., AND XU, B. Attention-based bidirectional long short-term memory networks for relation classification. In *Proc. of ACL* (2016).

[63] ZHU, Z., AND DUMITRAS, T. Featuresmith: Automatically engineering features for malware detection by mining the security literature. In *Proc. of CCS* (2016).

Appendix

A Model Parameters

The have tested the model performance with respect to different parameter settings for the NER model and the RE model. Figure 15 shows the RE model performance under different position embedding dimensions. Figure 16 to Figure 18 show the NER model performance under different batch sizes, the number of network layers, and the number of epochs. The result shows that our model is not very sensitive to parameter settings.

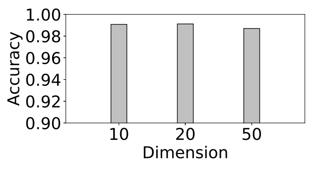


Figure 15: Position embed dim. vs. RE accuracy.

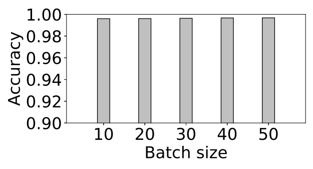


Figure 16: Batch size vs. NER accuracy.

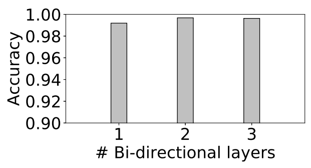


Figure 17: # Bi-directional layers vs. NER accuracy.

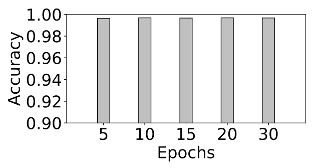


Figure 18: # Epochs vs. NER accuracy.