

# VulCPE: Context-Aware Cybersecurity Vulnerability Retrieval and Management

Yuning Jiang, Feiyang Shang, Freedy Tan Wei You, Huilin Wang, Chia Ren Cong, Qiaoran Meng, Nay Oo, Hoon Wei Lim, and Biplab Sikdar

**Abstract**—The dynamic landscape of cybersecurity demands precise and scalable solutions for vulnerability management in heterogeneous systems, where configuration-specific vulnerabilities are often misidentified due to inconsistent data in databases like the National Vulnerability Database (NVD). Inaccurate Common Platform Enumeration (CPE) data in NVD further leads to false positives and incomplete vulnerability retrieval. Informed by our systematic analysis of CPE and CVE details data, revealing more than 50% vendor name inconsistencies, we propose VulCPE, a framework that standardizes data and models configuration dependencies using a unified CPE schema (uCPE), entity recognition, relation extraction, and graph-based modeling. VulCPE achieves superior retrieval precision (0.766) and coverage (0.926) over existing tools. VulCPE ensures precise, context-aware vulnerability management, enhancing cyber resilience.

**Index Terms**—Data Inconsistency, Vulnerability Management.

## I. INTRODUCTION

Vulnerability management is a cornerstone of effective cyber defense, enabling organizations to prioritize and mitigate risks before attackers can exploit them. However, false positives (FPs) in vulnerability management predominantly stem from limitations in Common Platform Enumeration (CPE) [25] data utilized during the correlation process. This initial phase in vulnerability management matches deployed software against vulnerability databases maintained by NIST’s National Vulnerability Database (NVD) [26] and software vendors. Vulnerability scanners either extract granular software package data directly from vendor sources or rely on NIST’s CPE descriptions. While CPE aims to standardize software identification across vendors, empirical evidence suggests significant deficiencies in data accuracy and completeness [3, 9, 17, 19, 29].

Various open-source vulnerability management tools, such as *OpenCVE*, *OSV*, *cve-search*, *Trivy*, *CVEdetails* and *OpenVAS*, aim to improve vulnerability detection through database integration and search capabilities. *Trivy* [23] (container scanner) and *OpenVAS* [14] (network scanner) detect vulnerabilities by matching system components against *NVD* data. To enhance vulnerability data reliability, *OSV* [13], developed by Google, curates open-source vulnerability data using

ecosystem-based identifiers (e.g., npm, pip) instead of CPE, improving data accuracy and validation. *OpenCVE* [6] synchronizes CVE data from sources like NVD, MITRE, and RedHat, while *cve-search* [1] imports CVE and CPE data into a local MongoDB database, supporting fast searches and ranking vulnerabilities by CPE names. *CVEdetails* [8] supplements some missing CPE details but restricts API access to paid users, limiting its availability for programmatic queries. Proprietary tools like *Tenable* and *Fortinet* lack transparency, making direct comparisons difficult.

Despite these efforts, existing tools struggle with CPE inconsistencies, both FPs and false negatives (FNs), and incomplete mappings, which hinder vulnerability retrieval and integration [31]. Solutions relying on keyword searches or static CPE-based matching fail to address system configuration dependencies [32]. Tools such as *cve-search* and *OpenCVE* streamline retrieval but lack capabilities to mitigate FPs or support context-aware matching. Their reliance on manual processes further limits scalability and practicality in large-scale environments [24]. Meanwhile, the heterogeneous nature of software, hardware, and operating system (OS) configurations complicates the accurate mapping of vulnerabilities to affected assets [12, 30]. These structural limitations in CPE data representation frequently manifest as false positives in vulnerability detection systems, reducing the efficacy of automated security assessment protocols.

To address these challenges, we propose VulCPE, a framework that addresses these gaps by leveraging advanced techniques such as Named Entity Recognition (NER), Relation Extraction (RE), and graph-based modeling. Specifically, this work explores the following research questions:

**RQ1:** How do data inconsistencies in vulnerability databases affect retrieval accuracy?

**RQ2:** What role do complex system configurations play in determining vulnerability applicability?

**RQ3:** How can advanced techniques reduce false positives in vulnerability management to enhance cyber resilience?

We conducted a comprehensive analysis of the *NVD/CPE* and *CVEdetails* datasets to uncover prevalent inconsistency patterns. Our results show that 93.55% of *NVD* entries contain at least one valid CPE string. However, 81.40% of all defined CPE strings remain unused in the *NVD*, indicating significant underutilization of available configuration identifiers. Additionally, 14.56% of *NVD* entries rely on configuration-specific CPEs, which require parsing of logical AND/OR groupings. Naming inconsistencies were identified in 50.18% of vendor names used in CPEs within the official *NVD* database and

Y. Jiang, F. Shang, F. Tan Wei You, H. Wang, C. Ren Cong, Q. Meng, and B. Sikdar are with the National University of Singapore, Singapore (e-mail: yuning\_j@nus.edu.sg; hester03sfy@gmail.com; freedytan@u.nus.edu; hwang56@u.nus.edu; rencongchia@u.nus.edu; qiaoran@nus.edu.sg; bsikdar@nus.edu.sg).

N. Oo and H. W. Lim are with NCS Cyber Special Ops R&D, Singapore (e-mail: nay.oo@ncs.com.sg; hoonwei.lim@ncs.com.sg).

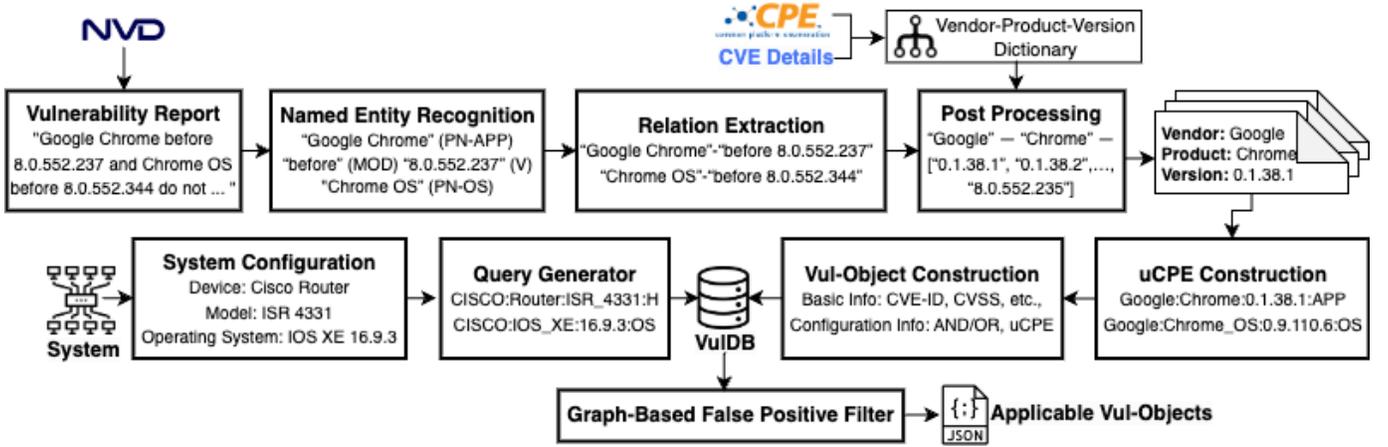


Fig. 1. VulCPE Architecture.

in 47.07% of vendor names extracted from *CVEdetails*, highlighting the need for standardization to enhance data usability.

Figure 1 illustrates the VulCPE framework. VulCPE employs NER and RE models to extract structured entities (vendor, product, version) from vulnerability reports and resolve inconsistencies in naming and formatting. Extracted data is standardized into a unified Common Platform Enumeration (uCPE) schema, which provides a hierarchical and logical representation of configurations. Logical relationships (*AND/OR*) and dependency structures (e.g., application software running on or alongside an OS) are modeled as directed graphs, enabling context-aware matching of vulnerabilities to system configurations. The system constructs two distinct graphs: a hierarchical graph of vulnerable configurations derived from uCPEs and a system configuration graph representing the system under investigation (SUI). Graph traversal techniques are used to match these configurations, ensuring precise vulnerability applicability assessments. Inconsistencies between configurations are detected using subgraph similarity measures, further reducing FPs.

Experimental results demonstrate the efficacy of VulCPE in two key areas. First, NER and RE models achieve state-of-the-art performance, with NER attaining a precision of 0.958 and recall of 0.975, and RE achieving a precision of 0.977 and recall of 0.914. Second, VulCPE significantly outperforms existing tools like *cve-search* and *OpenCVE* by achieving high retrieval coverage (0.926) and precision (0.766). Our manually labeled 5k ground-truth Common Platform Enumeration (CVE) reports for NER and RE model training and testing is released and available on IEEE DataPort [18].

The rest of this paper is organized as follows: Section II reviews vulnerability management systems and NER/RE applications in security. Section III analyzes *NVD*, *CPE*, and *CVEdetails* data inconsistencies. Section IV describes the VulCPE system architecture, NER/RE models, and uCPE formation. Section V addresses distributed deployment and resource management challenges. Section VI evaluates NER/RE performance and VulCPE's effectiveness in reducing FPs. Section VII presents conclusions and future directions.

## II. BACKGROUND AND RELATED WORK

### A. CPE, SCAP and SWID

The NIST Interagency Report 8085 outlines guidance for using Software Identification (SWID) tags to create standardized *CPE* names [33]. SWID tags, compliant with ISO/IEC 19770-2, enable accurate software identification across asset management and cybersecurity applications [34].

*CPE* functions as a dictionary for vulnerable products within the NIST Security Content Automation Protocol (SCAP) 1.2 standard. Each *CPE* entry includes type, vendor, product, and version information. For example, "*cpe:2.3:o:cisco\_xe:3.13.2as:::~::~\**" indicates an operating system (o) from vendor "*cisco*" with product "*ios\_xe*" version "*3.13.2as*". According to *NVD* [28], vulnerability configurations are classified as: (1) Basic Configuration with a single node holding one or more *CPE* names; (2) Running On/With Configuration containing multiple nodes with both vulnerable and non-vulnerable *CPE* names (Fig. 2); and (3) Advanced Configuration with multiple nodes and complex sets of *CPE* names. In this paper, we refer to both Running On/With and Advanced Configurations as Configuration-Specific CPEs.

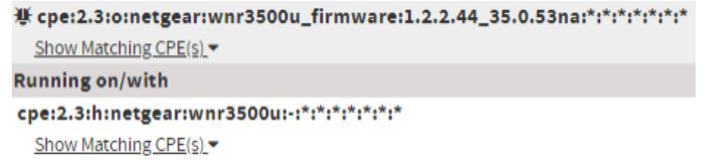


Fig. 2. Example of Running On/With configuration.

### B. Vulnerability Database Data Quality Analysis

Public databases like the CVE repository are commonly used in both research and commercial products for vulnerability analysis [2]. Yet, numerous recent investigations have highlighted the difficulties encountered with existing vulnerability databases, advocating for the creation of high-quality datasets [7] [21] [4] [11]. For example, Dong et al. [9] found significant inconsistencies in software version vulnerabilities reported between *CVE* and *NVD*, with only a fraction of

CVE summaries matching *NVD* entries accurately. Hong et al. [35] addressed the data inconsistencies and incorrectness in software names and versions, and emphasized the importance of identifying original vulnerable software. Li et al. [21] further carried out a comprehensive systematic mapping study focusing on the architecture and application of vulnerability databases. This investigation identifies dependencies on *NVD* and CVE databases, while also pointing out a significant shortfall in the existing vulnerability databases for their lack of detailed information and metadata, which poses a challenge to detecting vulnerabilities. Hong et al. [16] introduced a novel approach for database construction aimed at augmenting the scope of security patches. Their method involves correlating data from the *NVD* database with diverse sources such as repositories (e.g., GitHub), issue trackers (e.g., Bugzilla), and Q&A sites (e.g., Stack Overflow).

These findings emphasize the importance of developing methodologies [3, 15, 19] to enhance data consistency and completeness. Recent advancements in natural language processing [19, 20], machine learning [31] and graph-based [10] methods showed potential in extracting useful information from unstructured vulnerability reports. However, the quality of the trained data remains uncertain, which increases the challenges of applying these models in practical settings.

### C. NER and RE in Security Domains

Security vulnerability reports typically contain critical information such as software names, versions, and steps to reproduce the issue. Chaparro et al. [5] employed three distinct approaches, namely regular expressions, heuristics, and machine learning, to extract key elements from bug reports, including observed behavior, expected behavior, and steps to reproduce. In the context of vulnerability data, Semfuzz [36] utilized regular expressions to extract software version details from CVE entries, while VIEM [9] applied NER and RE techniques to extract software names and versions from vulnerability reports in six databases (e.g., *NVD*, ExploitDB and SecurityFocus). VERNIER [31], also based on NER, was designed to automatically extract software names from unstructured Chinese and English vulnerability reports and to measure inconsistencies in software names across nine mainstream databases (e.g., CVE, *NVD* and CNNVD). This method also used a reward-punishment matrix to detect incorrect software names, aiming to improve database accuracy.

Nevertheless, these existing solutions primarily focus on extracting software names and versions independently, without fully addressing the contextual relationships between vendor, product and version. This results in a fragmented understanding of vulnerabilities, which can lead to inaccurate retrieval and misidentification of relevant vulnerabilities in critical systems. Our work addresses this gap by utilizing *CPE* standards in combination with advanced NER and RE techniques to construct a unified, contextual representation of vendor, product, and version information. This graph-based uCPE structure not only captures the relationships among these entities but also allows for sophisticated traversal and configuration matching, enabling more accurate and context-

aware vulnerability retrieval. In addition, we design a dedicated database schema optimized for storing and retrieving vulnerabilities based on the uCPE structure. This schema is tailored to efficiently support queries that involve complex configurations, ensuring that vulnerabilities can be retrieved accurately with minimized false positives and false negatives.

## III. DATA ANALYSIS

This section examines the structure and inconsistencies in *NVD* and *CPE* data, highlighting configuration-based *CPE* patterns and naming inconsistencies in vendors and products.

### A. Preliminary Data Analysis of *NVD/CPE* Entries

1) *The Usage of CPE in NVD CVE Entries*: We obtained JSON feeds containing 259,233 vulnerability data from 2002 to 31 Aug 2024 (inclusive) from the official *NVD* website [27]. We then filtered these *NVD* entries based on their last modified date and excluded vulnerabilities marked as “*Rejected*” by the *NVD*, which leads to 244,819 vulnerabilities. The *CPE* v2.3 Dictionary was manually downloaded from *CPE* [25] and we parsed in total 1,327,827 *CPE* strings for further analysis. We processed all *NVD* entries to extract *CPE*-formatted strings and their associated configuration attributes. Of these 244,819 reviewed vulnerabilities, 229,023 (93.55%) contained at least one valid *NVD-CPE* string. Subsequent analyses focused on this subset. We noticed that some *NVD-CPE* strings are not recorded in the official *CPE* dictionary. Meanwhile, 81.40% of the official *CPE* strings were never referenced in *NVD*, indicating a significant portion of unused metadata.

2) *Running On/With CPE Entries*: Our analysis found that 14.56% of *NVD* entries specify configuration-specific *CPEs*, exhibiting four key patterns: OS dependencies (e.g., Product A runs on OS B), Enabled Modules (e.g., Product X is vulnerable when Module Y is enabled), Cloud/Virtualization Environments (e.g., vulnerabilities arise when guest virtual machines impact the host system), and Network Configurations (e.g., vulnerabilities caused by specific firewall rules).

Table I summarizes these configuration-specific *CPE* patterns. We extracted the *CPE* type (a: applications, o: OS, h: hardware devices) and generated all possible Running On/With relationships using Cartesian technique to capture each directed pair.

TABLE I  
COUNTS OF DIFFERENT CONFIGURATION COMBINATIONS.

Vulnerable CPE	Running On/With CPE	Count
o	a	3,711
o	h	1,224,357
o	o	4,071
a	a	58,426
a	h	26,350
a	o	297,491
h	a	436
h	h	933
h	o	2,158

Several patterns emerge from these results. OS-hardware configurations are most common (1,224,357 instances), followed by application-OS dependencies (297,491 cases). Less

frequent but notable configurations include OS-application (3,711), hardware-hardware (933), and OS-OS (4,071) combinations, which may indicate layered systems like virtual machines. Another common pattern is “\_firmware” appearing in vulnerable CPE product names (see Table II), with 21.20% of all configurations (343,015 cases), with 99.92% involving OS-hardware device relationships. The “firmware” keyword appears across all three CPE types, with 99.6% classified as OSs, potentially complicating vulnerability assessment. Additionally, 80.88% of configurations share the same vendor for both vulnerable and configuration CPEs, suggesting vulnerabilities often occur within vendor-controlled ecosystems.

TABLE II  
EXAMPLES OF CPE NAMES CONTAINING “firmware”.

Part of Vulnerable CPE	Product
a	small_business_rv_router_firmware
h	jetnet5628g-r_firmware
o	ethernet_controller_e810_firmware

These findings highlight the critical role of configuration-based CPEs in vulnerability data usability by providing essential context. Delays in updating these configuration details can significantly hinder timely vulnerability management.

### B. Heuristics for Detecting Inconsistencies

In vulnerability databases such as NVD and CVEdetails, inconsistencies in vendor and product names present significant challenges for accurate vulnerability retrieval and analysis. Given the large scale of vendor and product entries in these databases, manual identification of inconsistencies is impractical. We therefore filed a set of heuristics to detect and group potential name discrepancies for further validation. These heuristics address key patterns of variation observed.

Inconsistencies in *vendor* and *product* names are quantified as a pairwise divergence metric, where  $\text{sim}(\text{name}_1, \text{name}_2)$  denotes a similarity function, such as Levenshtein or Cosine similarity, calculated using:

$$\Delta(\text{name}_1, \text{name}_2) = 1 - \text{sim}(\text{name}_1, \text{name}_2). \quad (1)$$

An inconsistency is detected if the discrepancy is larger than a predefined similarity threshold  $\tau$ .

Define  $P(V)$  as the product set of vendor  $V$ , with  $P_{\text{norm}}(V) = \{\text{norm}(p) \mid p \in P(V)\}$ .  $\text{norm}$  is short for normalize. Shared Product Ratio (SPR) is:

$$\text{Sim}_{\text{prod}}(V_1, V_2) = \frac{|P_{\text{norm}}(V_1) \cap P_{\text{norm}}(V_2)|}{|P_{\text{norm}}(V_1) \cup P_{\text{norm}}(V_2)|}. \quad (2)$$

Pairwise heuristics require  $\text{Sim}_{\text{prod}}(V_1, V_2) \geq \theta_p$  (e.g., 0.5).

All the heuristics apply to inconsistency detection in vendor names. Meanwhile, the first heuristic (Format Variations) is also applied to detect inconsistencies in product names. In these cases, the product similarity condition ( $\text{Sim}_{\text{prod}} \geq \theta_p$ ) is replaced with vendor similarity ( $\text{Sim}_{\text{vendor}}$ ), defined as:

$$\text{Sim}_{\text{vendor}}(P_1, P_2) = \begin{cases} 1 & \text{if } \text{vendor}(P_1) = \text{vendor}(P_2). \\ 0 & \text{otherwise.} \end{cases} \quad (3)$$

For example, product names like *Windows 10* and *windows-10* from the same vendor (Microsoft) would be flagged as inconsistent under the Format Variations rule.

(1) Format Variations detects character-level differences in capitalization, punctuation, or special characters.

$$\Delta_{\text{format}}(V_1, V_2) = \begin{cases} 1 & \text{if } \text{norm}(V_1) = \text{norm}(V_2). \\ 0 & \text{otherwise.} \end{cases} \quad (4)$$

Inconsistency:  $\Delta_{\text{format}} = 1 \wedge \text{Sim}_{\text{prod}} \geq \theta_p$ . E.g., “*Microsoft Corp*” and “*microsoft-corp*”.

(2) Spelling Errors detect inconsistencies due to potential spelling or typographical errors in vendor names using edit distances. This is only applied to vendor names that share the same first letter, based on the linguistic observation that typographical errors rarely affect the initial character of a word. Let  $d_L(s_1, s_2)$  be the Levenshtein distance. For vendors with  $|\text{norm}(V_1)| \geq m$  and  $|\text{norm}(V_2)| \geq m$ , where  $m$  is a minimum length threshold (e.g.,  $m = 5$ ), define:

$$\text{Sim}_{\text{edit}}(V_1, V_2) = 1 - \frac{d_L(\text{norm}(V_1), \text{norm}(V_2))}{\max(|\text{norm}(V_1)|, |\text{norm}(V_2)|)} \quad (5)$$

$$\Delta_{\text{spelling}}(V_1, V_2) = \begin{cases} 1 & \text{if } \text{Sim}_{\text{edit}}(V_1, V_2) \geq \tau. \\ 0 & \text{otherwise.} \end{cases} \quad (6)$$

Inconsistency:  $\Delta_{\text{spelling}} = 1 \wedge \text{Sim}_{\text{prod}} \geq \theta_p$ , with  $\tau = 0.8$ . E.g., “*Microsoft*” and “*Microsfot*” have  $\text{Sim}_{\text{edit}}$  as 0.89.

(3) Substring Matches detect prefixes, suffixes, or substrings embedded within longer names, defined as:

$$\Delta_{\text{string}}(V_1, V_2) = \begin{cases} 1 & \text{if } \text{norm}(V_1) \subset \text{norm}(V_2) \vee \text{norm}(V_2) \subset \text{norm}(V_1). \\ 0 & \text{otherwise.} \end{cases} \quad (7)$$

Inconsistency:  $\Delta_{\text{string}} = 1 \wedge \text{Sim}_{\text{prod}} \geq \theta_p$ . E.g., “*Apache*” vs. “*Apache Software Foundation*”.

(4) Product Name as Vendor Name flags instances where products are referenced instead of vendors, defined as:

$$\Delta_{\text{prod}}(V) = \begin{cases} 1 & \text{if } \exists V' \neq V : \text{norm}(V) = \text{norm}(P) \wedge P \in P_{\text{norm}}(V'). \\ 0 & \text{otherwise.} \end{cases} \quad (8)$$

E.g., “*Windows*” instead of “*Microsoft*”.

(5) Shared Product Names identify cases where multiple vendors are linked to the same product, defined as:

$$\Delta_{\text{shared}}(V_1, V_2) = \begin{cases} 1 & \text{if } \text{Sim}_{\text{prod}}(V_1, V_2) \geq \theta_{\text{high}}. \\ 0 & \text{otherwise.} \end{cases} \quad (9)$$

E.g., “*Sun Microsystems*” and “*Oracle*” have post-acquisition overlap  $\theta$  as 0.8. For Shared Product Names, the SPR is defined as such:

$$\text{Sim}_{\text{prod}}(V_1, V_2) = \frac{|P_{\text{norm}}(V_1) \cap P_{\text{norm}}(V_2)|}{\min(P_{\text{norm}}(V_1), P_{\text{norm}}(V_2))}. \quad (10)$$

This is to account for cases where a smaller company has been acquired by a larger company, where the smaller company has much fewer products.

These heuristics serve as a foundational approach to detecting potential naming discrepancies that are then manually verified. For example, names such as “*heimdal*”, “*heimdalsecurity*” and “*heimdal\_project*” can be grouped and reviewed to determine whether they represent the same entity. If confirmed, they are treated as naming inconsistencies and standardized. An additional layer of validation is integrated by analyzing

TABLE III  
COMMON INCONSISTENCY PATTERNS IN VENDOR NAMING

Category	Shared Product Ratio $\geq 0.5$						Other Categories
	Format Variations	Format Variations (Exclude Capital Letter Differences)	Spelling Errors	Acronym	Substring Matches	Product as Vendor Name	Shared Product Names Shared Product Ratio $\geq 0.8$
Possible	29664 (59424)	8838 (17606)	133 (266)	7 (14)	458 (921)	615 (1242)	1594 (3728)
Confirmed	29664 (59424)	8838 (17606)	133 (266)	6 (12)	437 (880)	615 (1242)	1594 (3728)

<sup>1</sup> “Possible” groupings are generated by heuristics and validated as “Confirmed” inconsistencies through manual verification.

<sup>2</sup> Numbers outside parentheses represent unique vendor groups, while those inside denote associated names.

<sup>3</sup> Shared Product Ratio filters vendors sharing products, reducing false positives and refining heuristic groupings for manual verification.

shared product associations and cross-referencing external sources. Manual verification remains essential to distinguish true inconsistencies from cases where minor differences indicate distinct entities, such as separate firmware versions.

### C. Inconsistency Analysis

1) *Inconsistencies in Vendor Data:* Our analysis extended the initial dataset of 229,023 CVEs and associated 32,773 CPEs by incorporating 35,458 vendor-product-version pairs extracted from *CVEdetails*, a publicly available catalog of vendor and product information. We identified only 153 exact matched vendor names in *CPE* and *CVEdetails* when no normalization applies. This leads to a large set (67,925) of vendor names to be processed and standardized.

Our enhanced pipeline significantly extends the work of [3], which identified 1,835 inconsistent vendor names across 871 groups. In contrast, our method uncovered 65,482 inconsistent name instances grouped into 32,420 vendor clusters, as summarized in Table III. Format variations were the most common inconsistency, affecting 29,664 unique vendor groups and 59,424 name instances. These were primarily resolved through case folding, special character normalization, and token re-ordering. Such variations often arise from differing formatting conventions between *CPE* and *CVEdetails*, particularly in the use of capitalization, which can impair retrieval accuracy in case-sensitive systems. Excluding case-related issues, 8838 groups (17606 instances) still exhibited format variations due to other formatting differences. Other inconsistency patterns, including spelling errors, acronyms, sub-string matches, and instances where product names are mistakenly labeled as vendors, are analyzed separately in the subset that excludes format variations.

We observed FP pairing from acronym and substring matches, which were flagged during manual validation. To mitigate such errors, we integrated a Shared Product Ratio (SPR) threshold (Equation 2) as a validation heuristic. Vendor name pairs with an  $SPR \geq 0.5$  were flagged as potential matches, and those with an  $SPR \geq 0.8$  exhibited strong semantic coherence, often reflecting genuine aliasing. This filtering mechanism significantly improved precision by reducing the manual validation workload while maintaining high recall. The resulting Shared Product Names category included 1,594 confirmed vendor groups (3,728 name instances).

An important meta-level insight is that while format-based inconsistencies dominate quantitatively, the qualitative complexity and verification cost of semantic inconsistencies

(spelling, acronyms, substrings) are substantially higher. These patterns are more likely to propagate errors in downstream tasks such as vulnerability resolution, threat attribution, or software inventory reconciliation.

2) *Inconsistencies in Product Data:* In the analysis of product naming inconsistencies, the first step involved addressing the vendor name discrepancies identified in the previous phase. To achieve this, we remapped vendor names to their most consistent forms, prioritizing the name associated with the highest number of CVEs. This approach was grounded in the assumption that the vendor name linked to the greatest number of CVEs is the most widely accepted representation.

Product naming inconsistency analysis focused on the format variation heuristic. This heuristic effectively addressed inconsistencies arising from minor character formatting differences, such as underscores versus hyphens, while minimizing the need for manual validation. By prioritizing format variations, our analysis reduced FPs caused by similar product names across unrelated vendors. Among 225,192 unique products, the format variation heuristic identified 138,722 instances consolidated into 68,746 product groups, and hence 700,26 discrepancies primarily due to minor formatting issues. These findings emphasize the importance of standardized naming conventions to ensure consistency. Without such conventions, errors in vendor names propagate to product names, compounding inconsistencies and undermining data integrity.

3) *Impact of Data Inconsistency on Vulnerability Retrieval:* Approximately 48.67% (33,062) of the 67,925 vendor names exhibit inconsistencies, with 65,482 entries consolidated into 32,420 standardized names. For vendor names from the *CPE* dataset and *CVEdetails*, they each contains 16,444 (50.18%) and 16,697 (47.07%) inconsistencies. Moreover, even just within the consistent vendors, 70,026 product names (31.09% of 225,192) are affected by formatting variations.

Naming inconsistencies significantly hinder vulnerability retrieval by disrupting mappings between vulnerabilities and affected systems. Misaligned entries lead to incomplete assessments, where vulnerabilities are either overlooked or incorrectly associated. Such discrepancies delay patch identification and deployment, increasing the exposure window and the risk of exploitation. Moreover, the cumulative effect of these inconsistencies across large datasets can compound the risks, leading to widespread security gaps that are harder to detect and manage, as also discussed in works [3, 17, 31].

The analysis highlights that resolving inconsistencies requires scalable approaches to standardize naming conventions and enforce consistency across datasets. Automated normal-

ization techniques, cross-database validation, and metadata enrichment can improve data integrity, enabling more effective vulnerability identification, prioritization, and mitigation.

#### IV. METHODOLOGY OF VULCPE

This section provides an overview of VulCPE, detailing its architecture and key components designed for configuration-aware vulnerability retrieval and management.

##### A. Overview of VulCPE

The VulCPE architecture, illustrated in Fig. 1, processes vulnerability data to extract, standardize, and map system configurations for precise vulnerability retrieval.

The workflow begins with the **Data Pre-Processor**, which normalizes raw inputs from sources like *NVD* and *CVEdetails*, to ensure standardized data for downstream modules.

The **Named Entity Recognition (NER) Module** extracts cybersecurity-specific entities, including product names, versions, and types, from unstructured text. By leveraging domain-specific rules and configurations, the module ensures extracted entities reflect real-world system configurations.

The **Relation Extraction (RE) Module** maps relationships between recognized NER entities, such as product-version pairs, to enable precise configuration modeling.

Subsequently, the **Post Processing Module** comprises two key steps. First, the **Vendor & Product Separator** resolves vendor-product mappings using predefined heuristic rules and string similarity metrics, ensuring consistency with our canonical dictionaries. Next, with the processed vendor and product, the **Version Converter** translates complex version descriptors (e.g., “up to”, “before”) into normalized ranges based on datasets such as *NVD*. This step ensures consistency of vulnerable product versions across vulnerability sources.

**uCPE Generator** consolidates extracted product, version, and type data into hierarchical configurations, enabling interoperability and precise vulnerability-configuration mapping.

The **Vulnerability Database Constructor** structures processed data into a graph-based database  $G = (N, E)$ , where nodes ( $N$ ) represent entities (e.g., uCPE configurations) and edges ( $E$ ) capture relationships (e.g.,  $e_{AND}$ ,  $e_{OR}$ ) among components. This database facilitates efficient querying and supports configuration-aware vulnerability assessments.

**False Positive Filter** employs graph-based matching to refine vulnerability-configuration mappings. The system configuration graph ( $G_{sys}$ ) and vulnerability graph ( $G_{vul}$ ) are traversed to evaluate matches based on logical dependencies.

##### B. Named Entity Recognition

NER module extracts structured entities, namely *vendor* ( $v_i$ ), *product* ( $p_i$ ), *version* ( $ver_i$ ), and *type*  $t_i$  from unstructured vulnerability reports. Let  $T$  represent the text of a report. The extraction process is formally defined as:

$$\text{NER}(T) = \{(v_i, p_i, ver_i, t_i) \mid v_i, p_i, ver_i, t_i \in T\}. \quad (11)$$

Our NER model is built on RoBERTa [22], chosen for its ability to capture complex contextual relationships. Input text is tokenized into both word-level and sub-word-level

units, ensuring compatibility with out-of-vocabulary terms and multi-token entities. Each token is embedded into a dense vector representation, incorporating positional and sub-word-level embeddings. This approach effectively handles complex version formats with alphanumeric characters and punctuation (e.g., “v1.0.2-alpha”) and multi-token product names (e.g., “Google Chrome before 8.0.552.237”).

After initial embedding, tokens are further processed through self-attention layers, enabling the model to assign labels to tokens. The primary label set includes *Product Name* ( $PN$ ), *Modifier* ( $MOD$ ), *Version* ( $V$ ), and *Others* ( $O$ ). For instance, the previous is assigned “Google” ( $B$ - $PN$ ), “Chrome” ( $I$ - $PN$ ), “before” ( $B$ - $MOD$ ) and “8.0.552.237” ( $V$ ).

The model further integrates a domain-specific gazetteer derived from *CVEdetails* [8], containing vendor names, product names, and version ranges. This gazetteer is incorporated into a post-processing step to validate and adjust predictions using heuristic rules. For example, if the model labels “Internet” and “Explorer” as separate entities, the gazetteer merges them into “Internet Explorer” under a single  $PN$  label. This hybrid approach combines RoBERTa’s probabilistic predictions with deterministic rule-based corrections.

The NER module also captures product types (e.g., application, hardware, or OS). Using the same tokenizer and embeddings, extracted labels are concatenated with product type annotations. For instance, the earlier example is updated as “Google” ( $B$ - $PN$ - $APP$ ), “Chrome” ( $I$ - $PN$ - $APP$ ), “before” ( $B$ - $MOD$ ) and “8.0.552.237” ( $V$ ). This categorization ensures differentiations of product roles in system configurations.

##### C. Relation Extraction

The RE module identifies relationships between entities extracted by the NER module. With  $R$  represents the set of valid relationships, the relationship extraction process is formally defined as:

$$\text{RE}(v_i, p_i, ver_i, t_i) = \text{True} \iff (v_i, p_i, ver_i, t_i) \in R. \quad (12)$$

The RE model operates in two steps. It first groups modifiers and versions (e.g., “before 8.0.552.237”) together as ( $MOD\_V$ ). Then entities identified by the NER model are grouped into product-modifier-version ( $PN$ - $MOD\_V$ ) pairs. For each product ( $PN$ ), all associated modifiers and versions ( $MOD\_V$ ) within the same sentence are paired. For example, the vulnerability report results in the following four candidate pairs: “Google Chrome” with “before 8.0.552.237”; “Google Chrome” with “before 8.0.552.344”; “Google Chrome OS” with “before 8.0.552.237”; and “Google Chrome OS” with “before 8.0.552.344”. Each candidate pair is indexed based on its entity labels and converted into tokenized numerical representations, including token IDs, attention masks, and segment IDs. During inference, the RE model predicts the presence of a valid relationship ( $PN$ - $MOD\_V$ ) using logits generated from RoBERTa’s classification head, with “Y” indicating a valid relationship and “N” indicating its absence. If a valid relationship is detected, the model returns the corresponding ( $PN$ - $MOD\_V$ ) pairs.

#### D. Canonical Dictionary Creation

To standardize vendor, product, version and type data across heterogeneous sources, we construct a canonical dictionary of vendor-product-version-type pairs using CPE metadata utilized in NVD and a crawled CVEdetails dataset.

To resolve inconsistencies, a standardization function  $S(n^*, \mathcal{D})$  maps inconsistent names ( $n$ ) to a canonical form ( $n'$ ).  $\mathcal{D}$  is a dictionary of standardized names, using:

$$S(n^*) = \arg \max_{n' \in \mathcal{D}} d(n, n'). \quad (13)$$

The similarity between an extracted name (e.g., vendor  $v_i$  or product  $p_i$ ) and a canonical name  $n' \in \mathcal{D}$  is computed using similarity calculation, Levenshtein distance is used as an example:

$$\text{sim}(n, n') = 1 - \frac{\text{Lev}(n, n')}{\max(|n|, |n'|)}. \quad (14)$$

The canonical name is selected if the similarity exceeds a predefined threshold  $\tau$ :

$$n^* = \arg \max_{n' \in \mathcal{D}} \text{sim}(n, n') \quad \text{if } \text{sim}(n, n') \geq \tau. \quad (15)$$

NVD CPE strings are parsed into vendor, product, version, and type. Crawled CVEdetails data is flattened into a similar data frame, extracting vendor, product, and version lists. Next, we normalize both vendor and product names through a standardization process, which lowercases text, removes special characters, and standardizes whitespace. Inconsistency detection leverages heuristics from Section III-B: Format Variations, Spelling Variations, Acronyms, Substring Matches, Product Name as Vendor Name, and Shared Product Names. These heuristics are applied to both NVD and CVEdetails data, with consistent entries (via inner joins) forming the canonical dictionary and inconsistent ones (via left-anti joins) mapped to canonical names for traceability. Versions are grouped by normalized vendor-product pairs, combining unique versions from both sources.

In doing so, we obtain a canonical dictionary  $\mathcal{D}$  and separate mapping tables linking inconsistent names to canonical ones, supporting precise vulnerability retrieval.

#### E. Post Processing

The post-processing module processes two input sets or one of them: a set of extracted RE entries  $\mathcal{R} = \{\text{RE}_{\text{entry}_i} \mid i \in I\}$ , where each  $\text{RE}_{\text{entry}_i} = (v_i, p_i, \text{ver}_i, t_i)$ ; and a set of CPE match entries  $\mathcal{C} = \{\text{CPE}_{\text{entry}_j} \mid j \in J\}$ , where each  $\text{CPE}_{\text{entry}_j} = (v_j, p_j, \text{ver}_j, t_j)$ . We employ  $S(n^*, \mathcal{D})$  to standardize each  $\text{RE}_{\text{entry}_i}$  and  $\text{CPE}_{\text{entry}_j}$  to their canonical forms, as defined in Eq. (13). For vendor standardization,  $v_i$  is compared against  $V_{\text{canonical}} \subset \mathcal{D}$ , our canonical dataset of vendor names. After identifying  $v^*$ , the residual string is matched against products associated with  $v^*$  in  $\mathcal{D}$ . Product names are similarly standardized.

Version standardization converts textual version descriptions into mathematical constraints or discrete lists. Descriptions such as “*version 1.4 and earlier*” becomes “ $\leq 1.4$ ”, while “*not affected before version 5.0*” becomes “ $> 5.0$ ”. CPE-specific constraints, such as “*versionStartIncluding*”(≥) or “*versionEndExcluding*”(≤), are also parsed. Let  $v_{\text{desc}}$  be a

version description (from  $\text{ver}_i$  or  $\text{ver}_j$ ) and  $V_{\text{releases}}$  the set of available versions for a standardized vendor-product pair ( $v^*, p^*$ ). The version converter maps  $v_{\text{desc}}$  to a discrete list:

$$\text{List}(v_{\text{desc}}) = \{v_k \in V_{\text{releases}} \mid \text{cond}(v_k)\}, \quad (16)$$

Unlike [9], which assumes sequential versions, our approach supports non-sequential vendor releases. For example, “*Google Chrome before 8.0.552.344*” is converted to a list of actual releases: [0.1.38.1, 0.1.38.2, ..., 8.0.552.235].

The hybrid post-process combines entries from  $\mathcal{R}$  and  $\mathcal{C}$  to produce a set of normalized uCPE entries using canonical dictionary  $\mathcal{D}$ . If both  $\mathcal{R}$  and  $\mathcal{C}$  are empty, the process is skipped. When both  $\mathcal{R}$  and  $\mathcal{C}$  are non-empty, entries are aligned by computing similarity between standardized vendor-product pairs. If the similarity exceeds  $\tau$  and versions align, the CPE entry is prioritized. Unaligned entries are processed independently. Results are cached to avoid redundant computations.

#### F. Formation of uCPE

The uCPE schema addresses the challenges of complex relationships, such as “*Running On/With*” dependencies and nested configurations. A uCPE entry ( $\text{uCPE}_{\text{entry}}$ ) represents the foundational unit of vulnerability configuration, consisting a unique identifier, vendor name, product name, version, and product type (e.g., Application, OS, Hardware).

Configurations are modeled as subgraphs  $G_{\text{config}}$ , where  $N_{\text{uCPE}}$  represents nodes corresponding to individual components, and  $E_{\text{config}}$  defines the logical dependencies between components, using:

$$G_{\text{config}} = (N_{\text{uCPE}}, E_{\text{config}}). \quad (17)$$

Each edge in  $E_{\text{config}}$  represents either:

- *AND* relationships where components must coexist:

$$(\text{uCPE}_{\text{entry}_i} \wedge \text{uCPE}_{\text{entry}_j}) \rightarrow e_{\text{AND}}. \quad (18)$$

- *OR* relationships where at least one component suffices:

$$(\text{uCPE}_{\text{entry}_k} \vee \text{uCPE}_{\text{entry}_l}) \rightarrow e_{\text{OR}}. \quad (19)$$

Systems and vulnerabilities are modeled as graphs to represent their configurations and relationships.  $N_{\text{sys}}$  and  $N_{\text{vul}}$  are nodes representing  $\text{uCPE}_{\text{entry}}$  and their associated configurations.  $E_{\text{sys}}$  and  $E_{\text{vul}}$  are edges capturing logical relationships between uCPE entries or configurations, defined as:

$$G_{\text{sys}} = (N_{\text{sys}}, E_{\text{sys}}), \quad G_{\text{vul}} = (N_{\text{vul}}, E_{\text{vul}}). \quad (20)$$

Nodes in  $N_{\text{sys}}$  and  $N_{\text{vul}}$  represent either individual  $\text{uCPE}_{\text{entry}}$  elements or logical combinations. For example:

$$N_{\text{sys}} = \{\text{uCPE}_{\text{entry}_i}, (\text{uCPE}_{\text{entry}_j} \vee \text{uCPE}_{\text{entry}_k}), \dots\}. \quad (21)$$

For hierarchical relationships, the vulnerability graph  $G_{\text{vul}}$  for each CVE aggregates all uCPE configurations:

$$G_{\text{vul}} = \bigcup_{i=1}^n G_{\text{config}}(\text{uCPE}_{\text{entry}_i}). \quad (22)$$

### G. Database Construction and Retrieval

The database organizes our extracted information into three collections: *uCPE*, *Configurations*, and *Vulnerabilities*.

The *uCPE* Collection stores standardized vendor-product-version entries for interoperable vulnerability mapping, leveraging the canonical dictionary.

The *Configurations* Collection represents sub-graphs ( $G_{\text{config}}$ , Eq. (17)), with each entry containing a unique identifier (*config\_id*), logical relationship type ( $e_{\text{AND}}, e_{\text{OR}}$ ), and references to *uCPE* nodes, modeling hierarchical dependencies in the vulnerability graph  $G_{\text{vul}}$  (Eq. (22)).

The *Vulnerabilities* Collection links vulnerabilities to configurations via *config\_id*, including descriptions, CVSS scores, and exploitability metadata.

Two primary query types are implemented: one retrieves vulnerabilities based on CVE identifiers, while the other fetches vulnerabilities by matching specific product and version details. These queries leverage the hierarchical structure of  $G_{\text{sys}}$  and  $G_{\text{vul}}$ . This structure enhances VulCPE's precision and supports third-party scanners.

### H. Graph-Based False Positive Filtering

Our graph-based FP filtering technique leverages domain-specific cybersecurity knowledge to model relationships between vulnerabilities and assets. This approach incorporates configuration dependencies, logical relationships, and hierarchical asset structures, critical for precise vulnerability applicability assessments.

The applicability of a vulnerability node  $n_v \in N_{\text{vul}}$  to a system node  $n_s \in N_{\text{sys}}$  is determined by evaluating their hierarchical configurations.

For simple configurations without logical operators, the matching function evaluates whether the configuration graph of  $n_v$  is a subgraph of that of  $n_s$ :

$$\text{Match}(n_v, n_s) = \begin{cases} 1, & \text{if } G_{\text{config}}(n_v) \subseteq G_{\text{config}}(n_s), \\ 0, & \text{otherwise.} \end{cases} \quad (23)$$

For configurations involving logical operators, the matching function evaluates dependencies within  $E_{\text{config}}$ . Specifically:

$$\text{Match}(n_v, n_s) = \begin{cases} 1, & \text{if } \forall e_{\text{AND}} \in E_{\text{config}}(n_v), \text{ Match}(e_{\text{AND}}, n_s) = 1 \\ 1, & \text{if } \exists e_{\text{OR}} \in E_{\text{config}}(n_v), \text{ Match}(e_{\text{OR}}, n_s) = 1 \\ 0, & \text{otherwise.} \end{cases} \quad (24)$$

This matching process ensures that vulnerabilities are only applied when all AND conditions or any OR condition in the vulnerable configuration are matched by system configuration.

Further, the filtering process utilizes graph traversal to refine vulnerability applicability. Vulnerabilities ( $v$ ) and SUI are represented as vertices in  $G_{\text{vul}}$  and  $G_{\text{sys}}$ , enriched with logical dependencies. Algorithm 1 outlines the FP filtering procedure. If a match is found, the vulnerability is added to the set of applicable vulnerabilities ( $V_{\text{applicable}}$ ), as giving by:

$$V_{\text{applicable}} = V_{\text{vul}} - \{v \in V_{\text{vul}} \mid \text{Match}(v, n_s) = 1\}. \quad (25)$$

---

### Algorithm 1: Graph-Based False Positive Filtering

---

**Input:** System graph  $G_{\text{sys}} = (N_{\text{sys}}, E_{\text{sys}})$ , Vulnerability graph  $G_{\text{vul}} = (N_{\text{vul}}, E_{\text{vul}})$

**Output:** Set of applicable vulnerabilities  $V_{\text{applicable}}$

- 1 **Algorithm** *Graph-Based False Positive Filtering*:
- 2     Initialize  $V_{\text{applicable}} \leftarrow \emptyset$
- 3     **foreach**  $n_v \in N_{\text{vul}}$  **do**
- 4         **foreach**  $n_s \in N_{\text{sys}}$  **do**
- 5              $G_{\text{config}}(\text{vul}) \leftarrow \text{Traverse}(n_v, E_{\text{vul}})$
- 6              $G_{\text{config}}(\text{sys}) \leftarrow \text{Traverse}(n_s, E_{\text{sys}})$
- 7             **if**  $\text{Applicability}(G_{\text{config\_vul}}, G_{\text{config\_sys}})$  **then**
- 8                  $V_{\text{applicable}} \leftarrow V_{\text{applicable}} \cup \{n_v\}$
- 9     **return**  $V_{\text{applicable}}$
- 10 **Function**  $\text{Applicability}(G_{\text{config\_vul}}, G_{\text{config\_sys}})$ :
- 11     **if**  $\text{AND} \in E_{\text{config}}(\text{vul})$  **then**
- 12         **foreach**  $e_{\text{AND}} \in E_{\text{config}}(\text{vul})$  **do**
- 13             **if**  $\text{Match}(e_{\text{AND}}, G_{\text{config}}(\text{sys})) = 0$  **then**
- 14                 **return** False
- 15             **return** True
- 16     **if**  $\text{OR} \in E_{\text{config}}(\text{vul})$  **then**
- 17         **foreach**  $e_{\text{OR}} \in E_{\text{config}}(\text{vul})$  **do**
- 18             **if**  $\text{Match}(e_{\text{OR}}, G_{\text{config}}(\text{sys})) = 1$  **then**
- 19                 **return** True
- 20         **return** False
- 21 **Function**  $\text{Match}(element, G_{\text{config}}(\text{sys}))$ :
- 22     **return** 1 if  $element \in G_{\text{config}}(\text{sys})$ ; otherwise, 0.

---

## V. IMPLEMENTATION

We leverage several optimization strategies to enable VulCPE to handle large-scale vulnerability data while maintaining accuracy and minimizing computational overhead.

### A. Parallelization

Parallelization is implemented across multiple VulCPE modules to reduce processing time by distributing workloads. In the data pre-processing stage, text normalization and tokenization of vulnerability reports are executed concurrently using multi-threading, allowing independent processing of each report. Similarly, post-processing operations, including string similarity computations for standardizing vendor and product names, are parallelized across CPU cores, while database lookups for version conversions are batched to minimize I/O overhead. In the FP-filtering stage, graph-based subgraph isomorphism checks are distributed across multiple configurations.

### B. FP Handling

Our method is built upon [32] with three key improvements: Firstly, we utilize *uCPE-ID* than simply relying on the extracted textual information. Secondly, we use *NetworkX* to support graph implementation using *Python* to allow easier integration with the whole vulnerability pipeline. Thirdly, we enhance the efficiency of FP filtering by storing the graph

locally after its initial creation, and subsequently appending nodes upon the identification of new CVEs or Assets within the system. This approach significantly optimizes performance in terms of execution time. Empirical evidence from our experiments later in Section VI illustrates this improvement: the initial processing of 232 Assets requires approximately 25 minutes and 33 seconds. However, subsequent iterations demonstrate a marked reduction in execution time, involving only the verification of new assets rather than the comprehensive regeneration of the graph. Specifically, the addition of nodes for new assets incurs around 6.6 seconds per node, showcasing the efficiency of our optimized model in dynamically updating with minimal computational overhead.

### C. Incremental Updates

The graph-based vulnerability database is designed to support incremental updates, ensuring that new data can be integrated without requiring a full reconstruction. When new vulnerabilities or configurations are introduced, only the affected graph nodes and edges are updated, avoiding the computational expense of rebuilding the entire structure. This approach is also applied in the FP-filtering process, where the graph is modified incrementally upon the addition of new assets or vulnerabilities. Instead of reprocessing the entire dataset, filtering operations are restricted to newly introduced or updated nodes.

## VI. EXPERIMENTAL EVALUATION

This section presents a comprehensive experimental evaluation, with details on dataset, baseline models, evaluation metrics, and key implementation specifics. We focus on:

- RQ1: How effective is VulCPE in entity extraction and relation extraction compared to state-of-the-art approaches?
- RQ2: Can VulCPE be effectively applied to vulnerability retrieval in real-world settings?

### A. Experiment I: NER/RE Evaluation

1) *Dataset*: Previous NER datasets for vulnerability contexts [9] utilize simplistic annotation schemes (SN, SV, O) that inadequately capture nuanced entity boundaries and multi-token entities common in vulnerability data. Our review identified significant labeling gaps, necessitating a more comprehensive dataset for structured vulnerability descriptions.

We implemented a customized BIO format to label vulnerability reports, generating a ground-truth dataset for NER model training and validation. To enhance model performance, we expanded the NER label schema to include three product categories, replacing all B-PN/I-PN labels with categorized labels to improve *uCPE* matching and vulnerability retrieval.

From our dataset (Section III), we sampled 5,000 vulnerability descriptions (3,000 pre-2019 and 2,000 post-2019) for balanced temporal representation. Each description was tokenized and initially labeled using GPT-4o, though we observed relatively low accuracy, particularly for modifier (MOD) and version labeling. Consequently, two security researchers conducted manual reviews to ensure labeling accuracy.

To incorporate RE, we developed rules capturing relationships between product entities and their associated versions with modifiers. This approach identifies product-to-version relationships where modifiers define version applicability conditions (e.g., “before” a certain version or “fixed in” a particular release). We generated candidate pairs by linking product entities with version-modifier entities within the same context, assigning position indices for pairing. Additional contextual validation determined logical associations between the product and the version-modifier combination, with pairs labeled as valid (Y) or invalid (N).

2) *Evaluation Metrics*: Four main metrics are utilized to validate NER and RE models: (1) Accuracy is the fraction of correct predictions out of all predictions, offering a measure of overall correctness; (2) Precision is the ratio of correctly extracted entities and relations to the total identified, which minimizes false positives; (3) Recall is the proportion of correctly extracted entities and relations out of all relevant ones, which ensures true positives are included; (4) F1 Score is a harmonic mean of precision and recall, providing a balanced evaluation of accuracy and error rates.

3) *Implementation Details*: Our NER model employs the RoBERTa architecture via Hugging Face transformers, with labeled BIO format text split into training and testing sets (80-20) using a fixed random seed for reproducibility.

For RE, we utilize *RoBERTaForSequenceClassification* to identify entity relationships. Input sentences are preprocessed by tagging entities with custom tokens, then tokenized into IDs, masks, and segments to generate logits. Valid product-version pairs are extracted based on predictions.

4) *NER and RE Performance*: We evaluated our NER model, built on RoBERTa, against state-of-the-art baselines, including VERNIER [31] and VIEM [9]. VIEM results correspond to its best-performing configuration, incorporating transfer learning and gazetteer features, while VERNIER’s performance is reported for English-language vulnerability reports. We also included TinyLlama [37] which is a recent lightweight LLM that achieves competitive performance on token-level tasks. The RE evaluation of baseline models compares the set of predicted product-version relationships against the set of ground-truth relationships per sentence, using a greedy best-match approach with relaxed product aliasing and version matching. Table IV shows that our RoBERTa model with gazetteer achieved an accuracy of 98.56%, precision of 95.77%, recall of 97.54%, and an F1 score of 96.53%, demonstrating comparable performance to both baselines and outperforming simpler configurations such as RoBERTa without a gazetteer.

TABLE IV  
PERFORMANCE COMPARISON OF NER MODELS

Model	Accuracy	Precision	Recall	F1
RoBERTa (ours)	98.15%	96.31%	96.12%	96.22%
RoBERTa (Gaze) (ours)	98.56%	95.77%	97.54%	96.53%
[31] English Reports	99.8%	96.4%	96.9%	96.6%
[9] After Transfer	99.52%	94.85%	94.69%	94.77%
Tinyllama Zero-Shot	3.30%	5.41%	3.54%	4.03%
Tinyllama Few-Shot	20.41%	39.68%	26.14%	28.03%
Tinyllama Fine-tuned	37.02%	46.45%	43.82%	43.07%

For NER categorization across three categories (*APP*, *OS*, *HW*), we calculated both macro and weighted averages. As presented in Table V, the model achieved high recall for Applications (97.42%) and OSs (93.68%), while the performance for Hardware (79.01%) was lower due to the relatively smaller dataset and higher complexity in distinguishing hardware-related entities. The weighted average across categories reached 99.38% accuracy, demonstrating strong overall performance. The model achieved 99.38% weighted average accuracy, demonstrating robust overall performance, while the macro average (99.49% accuracy) confirmed balanced cross-category capability.

TABLE V  
PERFORMANCE OF NER CATEGORIZATION MODEL

Category	Accuracy	Precision	Recall	F1
Application	99.30%	96.46%	97.42%	96.94%
Operating System	99.78%	94.89%	93.68%	94.28%
Hardware	99.44%	83.97%	79.01%	81.41%
Macro Average	99.49%	91.77%	90.04%	90.88%
Weighted Average	99.38%	94.96%	95.03%	94.99%

Comparing RE model performance against VIEM [9], VIEM achieved slightly higher performance with ground-truth RE labels, while our model outperformed VIEM when using NER results as input. Tinyllama model’s near-zero recall (0.05% for zeros shot, 0.31% for few-shot, and 0.84% for fine tuning) reflects severe under-prediction, exacerbated by mismatches like “Oracle” vs. “oracle database”. RE model effectiveness depends significantly on NER output quality for entity identification and linking, with the pair generation process substantially influencing overall performance.

TABLE VI  
PERFORMANCE COMPARISON OF RE MODELS

Model	Accuracy	Precision	Recall	F1
Ours, G-truth as Input	97.41%	97.70%	91.44%	94.47%
[9] G-truth as Input	98.34%	97.81%	99.37%	99.09%
Ours, NER Result as Input	94.79%	94.71%	92.79%	93.74%
[9] NER Result as Input	90.44%	85.84%	99.64%	92.80%
Tinyllama Zero-Shot	0.05%	7.53%	0.05%	0.09%
Tinyllama Few-Shot	0.31%	20.41%	0.31%	0.60%
Tinyllama Fine-tuned	0.84%	51.59%	0.84%	1.65%

5) *Error Analysis*: We conducted thorough error analysis in our models and identified three main patterns. Our NER and RE models face challenges with complex product names and version mismatches. For example, in “Microsoft Word 2007 SP3, Office 2010 SP2”, “2007” is mislabeled as part of the product name (I-PN) instead of a version (B-V).

Ambiguity in platform vs. product classification is evident when “iOS” in “Newphoria Auction Camera for iOS” is misclassified as a product (B-PN) instead of a non-entity (O).

Product-version confusion occurs, as in date-based versions like “2017-02-12” in “Android for MSM before 2017-02-12” that cause boundary errors.

Heuristic post-processing rules partially mitigate these errors by reclassifying year-based identifiers (e.g., “2007”) as versions and normalizing complex version patterns, improving boundary detection. We also utilized context clues (e.g.,

prepositions like “for”) to distinguish platforms and by flagging common product name suffixes like “Edition” as I-PN, reducing misclassifications.

## B. Experiment II: Vulnerability Retrieval

1) *Dataset*: To simulate a real-world use-case scenario for our comparative analysis, we randomly selected and stored commonly used software packages within our testing environment. We then generated a system configuration file that comprised three distinct components: a network device segment consisting of 4 components, two virtual machines, one based on Linux OS and one based on Windows, with 46 and 22 components, respectively.

2) *Steps*: We queried the system’s configuration against multiple vulnerability databases: *NVD*, *cve-search*, *OpenCVE* and our proprietary database. This yields separate sets of vulnerabilities, denoted as  $V_{nvd}$ ,  $V_{cvesearch}$ ,  $V_{opencve}$  and  $V_{our}$ , respectively. A union set,  $V_{union}$ , is constructed from the individual sets to encompass all unique vulnerabilities identified across the databases. We did not involve *OSV*, *Security Database* and *CVEdetails*, due to different focus for *OSV* and limited accessibility for *Security Database* and *CVEdetails*. We further produced several sub-databases considering various query methods provided by *NVD API*, *cve-search* and *OpenCVE* in terms of keyword (exact) match and *CPE* match, following their official query instructions. For the latter, we use the *uCPE* metadata generated in our vulnerability pipeline as query tags.

A manual verification process is conducted on  $V_{union}$  to determine the applicability of each vulnerability to our system, involving a detailed review of vulnerability reports and matching identified vulnerabilities against the system configuration. Through the manual verification process, we establish a ground-truth dataset  $V_{gt}$ , representing the accurately identified vulnerabilities applicable to our system. We then compare  $V_{gt}$  against each database-specific vulnerability set (e.g.,  $V_{nvd}$ ,  $V_{cvesearch}$ ,  $V_{opencve}$ ,  $V_{our}$ ).

3) *Evaluation Metrics*: Validation of vulnerability retrieval performance involves calculating FP (or False Positives), FN (or False Negatives), TP (or True Positives), Retrieval Precision and Retrieval Coverage for each dataset. We then calculate the average of them. Here  $V_n$  denotes a database-specific vulnerability set and  $V_{gt}$  denotes a ground-truth dataset.

- FP: Vulnerabilities in  $V_n$  but not in  $V_{gt}$ .
- FN: Vulnerabilities in  $V_{gt}$  but not in  $V_n$ .
- TP: Vulnerabilities in both  $V_n$  and  $V_{gt}$ .
- Retrieval Precision:  $TP/(TP + FP)$ , the fraction of correctly identified vulnerabilities.
- Retrieval Coverage:  $TP/(TP + FN)$ , the fraction of actual vulnerabilities correctly identified.

4) *Results*: The results are summarized in Table VII. The baseline outcomes, displayed in Columns 2 to 5, illustrate the precision and coverage of vulnerability retrieval using various methods: exact matching of *NVD* keywords via the *NVD API*, keyword matching with localized *cve-search* database, and localized *OpenCVE*. Enhanced baseline results leveraging our *CPE* metadata tags as queries are detailed in Columns 6 to 8.

TABLE VII  
COMPARATIVE STUDY RESULTS OF VULNERABILITY RETRIEVAL

Metrics	Baseline Solutions				Improved Baseline with <i>CPE</i> Query			Ours
	NVD (keyword)	NVD (keyword exact)	cve-search (keyword)	<i>OpenCVE</i> (keyword)	NVD (CPE)	cve-search (product)	<i>OpenCVE</i> (CPE)	
Precision (LinuxVM)	0.143	0	0	0.288	0.875	<b>0.959</b>	0.269	0.949
Coverage (LinuxVM)	0.011	0	0	0.408	<b>0.951</b>	0.897	0.587	0.918
Precision (WinVM)	0.24	0.5	0	0	0.626	0.511	0.433	<b>0.667</b>
Coverage (WinVM)	0.015	0.005	0	0	<b>0.932</b>	0.624	0.237	0.879
Precision (Routers)	0.063	0.333	0	0	0.627	0.567	0.125	<b>0.683</b>
Coverage (Routers)	0.024	0.024	0	0	<b>0.980</b>	0.905	0.119	<b>0.980</b>
Precision (Average)	0.149	0.278	0	0.096	0.709	0.679	0.276	<b>0.766</b>
Coverage (Average)	0.017	0.010	0	0.136	<b>0.954</b>	0.809	0.314	0.926

Typically, vulnerability analyzers are limited to system configuration data and lack comprehensive configuration-based metadata for precise vulnerability identification. Incorporating *CPE* query data improved precision and coverage across all baseline databases, confirming our assumption that standardized metadata enhances retrieval accuracy. Our vulnerability pipeline achieved the highest average precision of 72.6%. In terms of coverage, our solution provided a good result of 92.6%, close to the highest coverage of 95.4% achieved by *NVD* when using our generated *CPE* metadata as query tags.

## VII. CONCLUSION

This paper presents VulCPE, a cybersecurity-focused framework that addresses critical challenges in vulnerability management, including data inconsistencies and false positives in existing vulnerability databases. By leveraging NER and RE techniques, VulCPE standardizes vendor, product, and version relationships into a uCPE schema. This approach enhances vulnerability retrieval by resolving inconsistencies, improving context-aware mapping, and enabling accurate applicability assessments across diverse and complex configurations.

Experimental studies demonstrated the efficacy of our proposed framework, showcasing better performance in terms of vulnerability retrieval precision and coverage compared to open-source baseline solutions (*NVD*, *cve-search*, and *OpenCVE*). Our proposed query generation mechanism expands vulnerability coverage across all baseline solutions. Moreover, our vulnerability pipeline achieves the highest precision (0.766) and coverage (0.926) in vulnerability retrieval, surpassing figures obtained using *NVD*, *cve-search*, and *OpenCVE*. These outcomes show the efficacy of our automated FP filtering mechanisms. Additionally, VulCPE’s NER and RE models outperform baseline approaches, with the NER model achieving 0.958 precision and 0.975 recall, and the RE model providing more accurate vulnerability-to-version mappings with precision of 0.977 and recall of 0.914.

Future work will focus on scaling VulCPE to enterprise environments of varying sizes and industries, exploring integrations with commercial vulnerability management systems, and testing whether more advanced LLMs (e.g., Llama and GPT-x) could be used as alternatives for NER and RE tasks.

## ACKNOWLEDGMENTS

We also thank the contributors who helped inconsistency analysis and pipeline development, especially Yong Zi Ren, Seng Chin Khoo and Lee Yu Yee Dominic.

## REFERENCES

- [1] Cve-search. <https://www.circl.lu/services/cve-search/>, 2025.
- [2] Sultan S Alqahtani. A study on the use of vulnerabilities databases in software engineering domain. *Computers & Security*, 116:102661, 2022.
- [3] Afsah Anwar, Ahmed Abusnaina, Songqing Chen, Frank Li, and David Mohaisen. Cleaning the nvd: Comprehensive quality assessment, improvements, and analyses. 2020.
- [4] Guru Bhandari, Amara Naseer, and Leon Moonen. Cve-fixes: automated collection of vulnerabilities and their fixes from open-source software. In *Proceedings of the 17th International Conference on Predictive Models and Data Analytics in Software Engineering*, pages 30–39, 2021.
- [5] Oscar Chaparro, Jing Lu, Fiorella Zampetti, Laura Moreno, Massimiliano Di Penta, Andrian Marcus, Gabriele Bavota, and Vincent Ng. Detecting missing information in bug descriptions. In *Proceedings of the 2017 11th joint meeting on foundations of software engineering*, pages 396–407, 2017.
- [6] Nicolas Crocfer and Laurent Durnez. Opencve: A cve alertig platform. <https://github.com/opencve/opencve>, 2025.
- [7] Roland Croft, Yongzheng Xie, and Muhammad Ali Babar. Data preparation for software vulnerability prediction: A systematic literature review. *IEEE Transactions on Software Engineering*, 49(3):1044–1063, 2022.
- [8] CVEdetails. Cvedetails. <https://www.cvedetails.com/>, 2025.
- [9] Ying Dong, Wenbo Guo, Yueqi Chen, Xinyu Xing, Yuqing Zhang, and Gang Wang. Towards the detection of inconsistencies in public security vulnerability reports. In *28th USENIX security symposium (USENIX Security 19)*, pages 869–885. USENIX Association, August 2019.
- [10] Dongdong Du, Xingzhang Ren, Yupeng Wu, Jien Chen, Wei Ye, Jinan Sun, Xiangyu Xi, Qing Gao, and Shikun

- Zhang. Refining traceability links between vulnerability and software component in a vulnerability knowledge graph. In *Web Engineering: 18th International Conference, ICWE 2018, Cáceres, Spain, June 5-8, 2018, Proceedings 18*, pages 33–49. Springer, 2018.
- [11] Sadegh Farhang, Mehmet Bahadır Kirdan, Aron Laszka, and Jens Grossklags. An empirical study of android security bulletins in different vendors. In *Proceedings of The Web Conference 2020*, pages 3063–3069, 2020.
- [12] Rohit Gangupantulu, Tyler Cody, Abdul Rahma, Christopher Redino, Ryan Clark, and Paul Park. Crown jewels analysis using reinforcement learning with attack graphs. In *2021 IEEE Symposium Series on Computational Intelligence (SSCI)*, pages 1–6. IEEE, 2021.
- [13] Google. Osv - ppen source vulnerability db and triage service. <https://google.github.io/osv.dev/>, 2025.
- [14] Greenbone. Greenbone openvas. <https://www.openvas.org/>, 2025.
- [15] Erik Hemberg, Matthew J Turner, Nick Rutar, and Una-May O’reilly. Enhancements to threat, vulnerability, and mitigation knowledge for cyber analytics, hunting, and simulations. *Digital Threats: Research and Practice*, 5(1):1–33, 2024.
- [16] Hyunji Hong, Seunghoon Woo, Eunjin Choi, Jihyun Choi, and Heejo Lee. xvdb: A high-coverage approach for constructing a vulnerability database. *IEEE Access*, 10:85050–85063, 2022.
- [17] Yuning Jiang, Manfred Jeusfeld, and Jianguo Ding. Evaluating the data inconsistency of open-source vulnerability repositories. In *Proceedings of the 16th International Conference on Availability, Reliability and Security*, pages 1–10, 2021.
- [18] Yuning Jiang, Feiyang Shang, and Wei You (Freedy) Tan. Labelled dataset for ner/re tasks in vulnerability reports. <https://dx.doi.org/10.21227/aggr-d448>, 2025.
- [19] Hyeonseong Jo, Jinwoo Kim, Phillip Porras, Vinod Yegneswaran, and Seungwon Shin. Gapfinder: Finding inconsistency of security information from unstructured text. *IEEE Transactions on Information Forensics and Security*, 16:86–99, 2020.
- [20] Hyeonseong Jo, Yongjae Lee, and Seungwon Shin. Vulcan: Automatic extraction and analysis of cyber threat intelligence from unstructured text. *Computers & Security*, 120:102763, 2022.
- [21] Xiaozhou Li, Sergio Moreschini, Zheyang Zhang, Fabio Palomba, and Davide Taibi. The anatomy of a vulnerability database: A systematic mapping study. *Journal of Systems and Software*, 201:111679, 2023.
- [22] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. Roberta: A robustly optimized bert pretraining approach. *ArXiv*, abs/1907.11692, 2019.
- [23] Aqua Security Software Ltd. Trivy documentation. <https://github.com/aquasecurity/trivy>, 2025.
- [24] Joshua Lubell and Timothy Zimmerman. Challenges to automating security configuration checklists in manufacturing environments. In *Critical Infrastructure Protection XI: 11th IFIP WG 11.10 International Conference, ICCIP 2017, Arlington, VA, USA, March 13-15, 2017, Revised Selected Papers 11*, pages 225–241. Springer, 2017.
- [25] MITRE. Common platform enumeration (cpe). <https://cpe.mitre.org/>, 2025.
- [26] National Institute of Standards and Technology (NIST). National vulnerability database (nvd). <https://nvd.nist.gov/vuln>, 2025.
- [27] National Institute of Standards and Technology (NIST). Nvd vulnerability data feeds. <https://nvd.nist.gov/vuln/data-feeds#RSS>, 2025.
- [28] National Institute of Standards and Technology. Vulnerability details in nvd. <https://nvd.nist.gov/vuln/vulnerability-detail-pages>, 2025.
- [29] OWASP. A proposal to operationalize component identification for vulnerability management. White paper, OWASP Foundation, 2022.
- [30] Yue Qin, Yue Xiao, and Xiaojing Liao. Vulnerability intelligence alignment via masked graph attention networks. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, pages 2202–2216, 2023.
- [31] Hongyu Sun, Guoliang Ou, Ziqiu Zheng, Lei Liao, He Wang, and Yuqing Zhang. Inconsistent measurement and incorrect detection of software names in security vulnerability reports. *Computers & Security*, 135:103477, 2023.
- [32] Daniel Tovarňák, Lukáš Sadlek, and Pavel Čeleda. Graph-based cpe matching for identification of vulnerable asset configurations. In *2021 IFIP/IEEE International Symposium on Integrated Network Management (IM)*, pages 986–991. IEEE, 2021.
- [33] David Waltermire and Brant Cheikes. Forming common platform enumeration (cpe) names from software identification (swid) tags. Technical report, National Institute of Standards and Technology, 2015.
- [34] David Waltermire, Brant A Cheikes, Larry Feldman, David Waltermire, and Greg Witte. *Guidelines for the creation of interoperable software identification (SWID) tags*. US Department of Commerce, National Institute of Standards and Technology, 2016.
- [35] Seunghoon Woo, Dongwook Lee, Sunghan Park, Heejo Lee, and Sven Dietrich. {VOfinder}: Discovering the correct origin of publicly reported software vulnerabilities. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 3041–3058. USENIX Association, August 2021.
- [36] Wei You, Peiyuan Zong, Kai Chen, XiaoFeng Wang, Xiaojing Liao, Pan Bian, and Bin Liang. Semfuzz: Semantics-based automatic generation of proof-of-concept exploits. In *Proceedings of the 2017 ACM SIGSAC conference on computer and communications security*, pages 2139–2154, 2017.
- [37] Peiyuan Zhang, Guangtao Zeng, Tianduo Wang, and Wei Lu. Tinyllama: An open-source small language model. *arXiv preprint arXiv:2401.02385*, 2024.

## APPENDIX

Fig. 3 illustrates the architecture of our NER module based on RoBERTa. The pipeline tokenizes input text and assigns BIO-format labels to each token, with enhanced categorization (e.g., *(B-PN-APP)*) appended post-processing to reflect entity roles such as applications, OS, or hardware.

Fig. 4 shows the RE module that identifies valid product-version relationships using token pair classification. Candidate *(PN-MOD\_V)* pairs are formed from NER output, tokenized with position encodings, and passed through a RoBERTa-based attention network to determine whether a valid relationship exists.

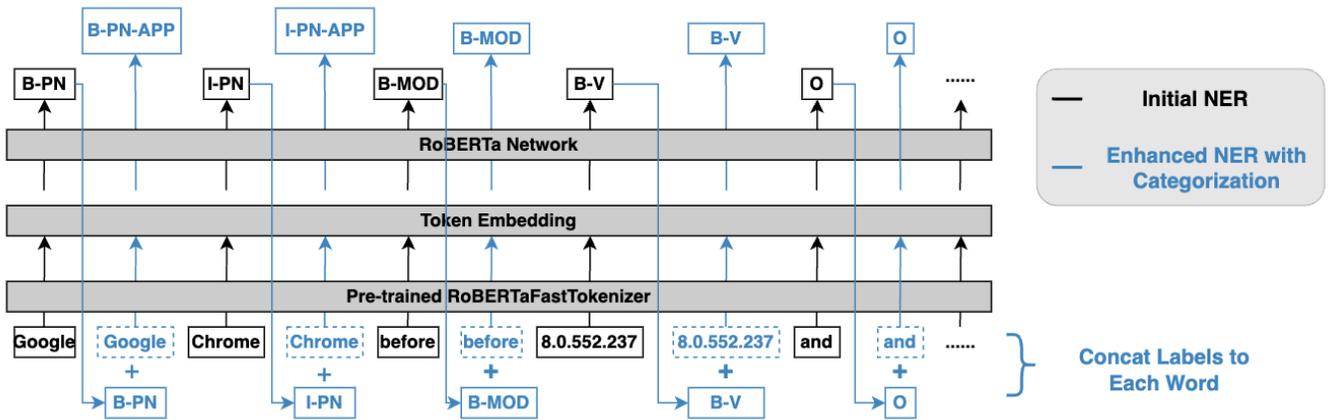


Fig. 3. Structure of Named Entity Recognition Module

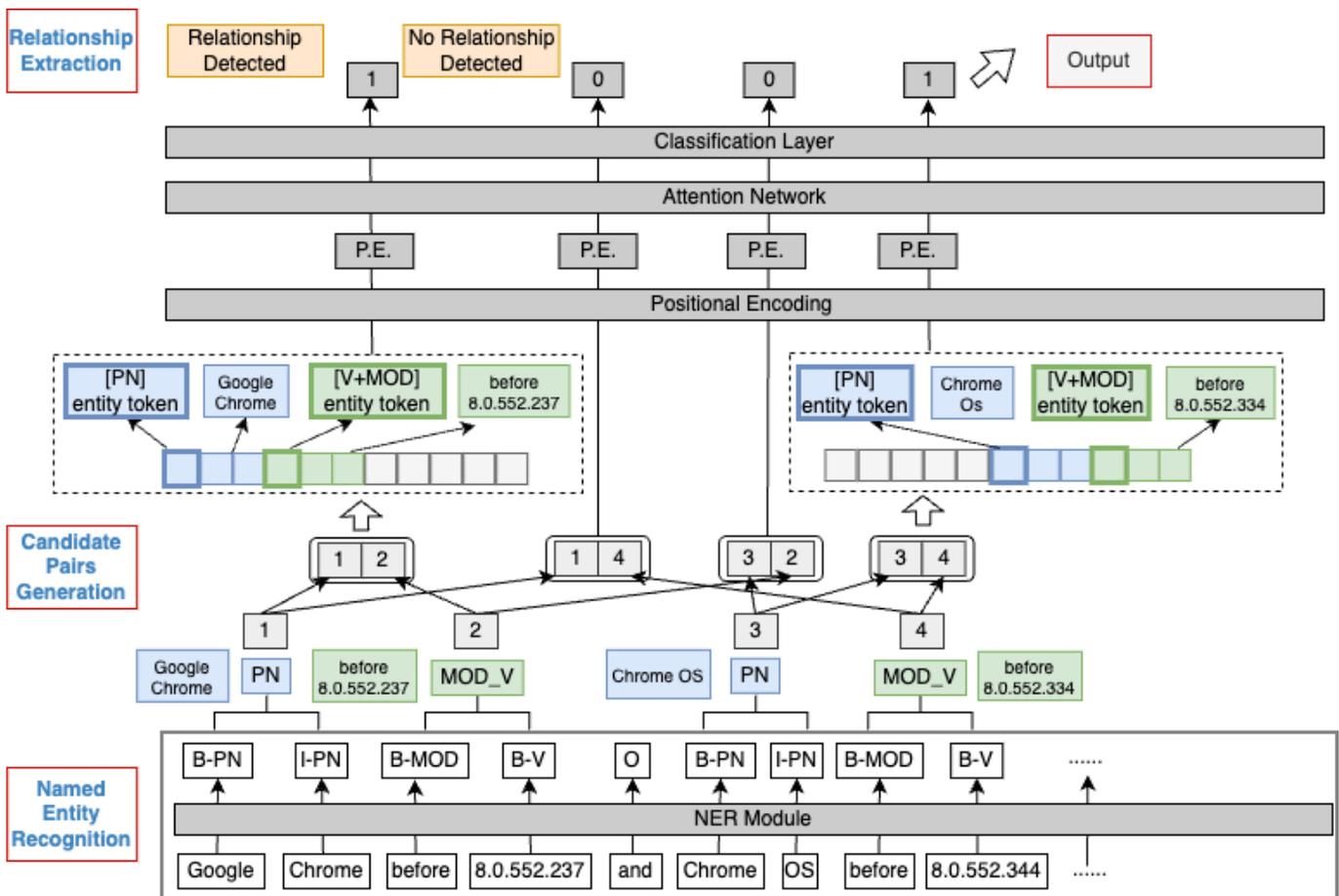


Fig. 4. Structure of Relation Extraction Module