IEEE Open Journal of the
Computer Society

Received 28 May 2024; revised 24 June 2024; accepted 1 July 2024. Date of publication 3 July 2024;
date of current version 2 August 2024. The review of this article was arranged by Associate Editor Peng Li.

Digital Object Identifier 10.1109/OJCS.2024.3422686

A Taxonomy for Python Vulnerabilities

FRÉDÉRIC C. G. BOGAERTS [1], NAGHMEH IVAKI [1] (Member, IEEE), AND JOSÉ FONSECA [2]

[1] Department of Informatics Engineering, University of Coimbra, CISUC, DEI, 3004-531 Coimbra, Portugal
[2] Department of Informatics Engineering, Polytechnic Institute of Guarda, University of Coimbra, CISUC, 6300-559 Guarda, Portugal

CORRESPONDING AUTHOR: FRÉDÉRIC C. G. Bogaerts (e-mail: fbogaerts@dei.uc.pt).

This work was supported by the FCT - Foundation for Science and Technology, I.P./MCTES through national funds (PIDDAC), within the scope of CISUC R&D Unit- UIDB/00326/2020 or project code UIDP/00326/2020 (2023.00319.BD).

ABSTRACT Python is one of the most widely adopted programming languages, with applications from web development to data science and machine learning. Despite its popularity, Python is susceptible to vulnerabilities compromising the systems that rely on it. To effectively address these challenges, developers, researchers, and security teams need to identify, analyze, and mitigate risks in Python code, but this is not an easy task due to the scattered, incomplete, and non-actionable nature of existing vulnerability data. This article introduces a comprehensive dataset comprising 1026 publicly disclosed Python vulnerabilities sourced from various repositories. These vulnerabilities are meticulously classified using widely recognized frameworks, such as Orthogonal Defect Classification (ODC), Common Weakness Enumeration (CWE), and Open Web Application Security Project (OWASP) Top 10. Our dataset is accompanied by patched and vulnerable code samples (some crafted with the help of AI), enhancing its utility for developers, researchers, and security teams. In addition, a user-friendly website was developed to allow its interactive exploration and facilitate new contributions from the community. Access to this dataset will foster the development and testing of safer Python applications. The resulting dataset is also analyzed, looking for trends and patterns in the occurrence of Python vulnerabilities, with the aim of raising awareness of Python security and providing practical, actionable guidance to assist developers, researchers, and security teams in bolstering their practices. This includes insights into the types of vulnerabilities they should focus on, the most exploited categories, and the common errors that programmers tend to make while coding that can lead to vulnerabilities.

INDEX TERMS Computing milieux, error handling and recovery, management of computing and information systems, reliability, software engineering, software/software engineering, software quality/SQA, security and protection, testing and debugging.

I. INTRODUCTION

Python has emerged as one of the most widely adopted programming languages, with applications spanning web development, data science, machine learning, and beyond. According to the TIOBE index [1], Python was ranked the most popular programming language in February 2024, with a market share of 15.16%. However, despite its popularity, Python is susceptible to security vulnerabilities that can compromise systems relying on it. Moreover, Python's versatility across multiple platforms (Windows, MacOS, Linux, Unix) has led to its widespread adoption, but also increases its attack surface. Addressing these vulnerabilities is crucial for ensuring the confidentiality, integrity, and availability of Python-based applications [2].

A. PROBLEM STATEMENT

Security vulnerabilities are flaws or weaknesses in software design, implementation, or operation that attackers can exploit to cause harm or gain unauthorized access [3].

In our previous research, we introduced a preliminary version of Vulnerability Attack and Injection Tool for Python (VAITP) [4], and we underscored the importance of having a comprehensive dataset of Python vulnerabilities, which can serve as a valuable resource for developers and researchers alike. Such a dataset, containing Python code samples encompassing both vulnerable and secure instances, is essential to train both practitioners and artificial intelligence (AI) models capable of detecting, injecting, exploiting, and patching vulnerabilities.

© 2024 The Authors. This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

The characterization and classification of security vulnerabilities play pivotal roles: they empower developers to proactively create secure code, assist quality assurance teams in designing effective tests to identify and rectify vulnerable code, enable the vulnerability detection and injection communities to develop effective tools and facilitate the broader security community in enhancing the training of security teams, among others. However, these tasks are not trivial and require a comprehensive and systematic understanding of each vulnerability's nature, characteristics, impact, and mitigation, which can only be found incomplete and spread across the web in several repositories. Moreover, these tasks are challenging, as they involve dealing with a large and diverse set of vulnerabilities, each with its own specificities and complexities.

While established taxonomies such as Orthogonal Defect Classification (ODC) [5], Common Weakness Enumeration (CWE) [6], and Open Worldwide Application Security Project (OWASP) Top 10 [7], provide valuable taxonomies for understanding and classifying vulnerabilities, our research underscores the necessity for a more comprehensive approach to classify Python vulnerabilities. In addition to these taxonomies, in the classification of our dataset, we introduce categories based on the root cause and specificities of vulnerabilities (e.g., CVE-2019-17526 refers to a "Command Injection" due to insufficient "Input Validation and Sanitization" of data that is passed to the *eval* function call), providing a more detailed perspective on the nature of security vulnerabilities in Python, cross-referencing with these well-established taxonomies, to narrow the divide between conventional categorizations and the dynamically evolving threat landscape that is specific to the Python programming language, its packages,[1] and libraries.[2] Thus, this approach improves the understanding of Python vulnerabilities and provides a valuable and comprehensive resource for developers, researchers, and security teams alike.

B. CONTRIBUTIONS

In this article, we present a comprehensive dataset of 1026 publicly known Python vulnerabilities, classified according to several existing taxonomies, such as ODC [5], CWE [6], and OWASP Top 10 [7].

In addition, we constructed a vulnerability taxonomy, comprising 10 custom categories and 41 subcategories based on vulnerability type and specificities.

Our study began with an examination of multiple online repositories, such as CVE Details [8], CVE.org [9], Snyk [10], NVD [11], and CVE Fixes [12],[3] as well as security tools, such as Bandit [13], Semgrep [14], and Sonar Lint [15]. This comprehensive review allowed us to identify known vulnerabilities in Python and construct a dataset for further analysis. Additionally, we supplemented the dataset with data generated by AI models in an AI-in-The-Loop (AIiTL)[4] approach, wherein human confirmation verified the accuracy of the data, mitigating the risk of dataset contamination with possible AI hallucinations[5] [16], [17].

Our work offers several potential contributions. The expert-reviewed Python vulnerability code blocks can help security teams deepen their understanding of vulnerability characteristics, serving as a valuable resource for training. Additionally, our code dataset enables further testing of vulnerability detection and injection tools.

We also provide a website that allows users to access and explore the dataset interactively and in a user-friendly manner [18] and contribute to the completeness and correctness of the dataset's information.

Finally, we present an exploratory data analysis (EDA) of Python-related vulnerabilities, including trend analysis and descriptive statistics, to provide insights into the current landscape of these vulnerabilities. The data for the EDA was generated using Python scripts, which are also available on the project's website.

The current work also lays the foundation for future endeavors we intend to pursue, such as testing and evaluating security tools, creating and refining AI models for vulnerability detection, injection, and exploitation, and maintaining VAITP designed to scan, inject, and exploit vulnerabilities. The overarching goal of this research is to foster a deeper understanding of Python vulnerabilities, provide practical tools and resources, and ultimately contribute to the development of more secure and resilient Python applications.

C. OUTLINE OF THE ARTICLE

Section II reviews related work on vulnerability and fault classification. Section III describes the methodology, including data sources, web and code scraping processes, and classification criteria. Section IV presents and analyzes our dataset, covering aspects such as the quantity and distribution of vulnerabilities, trends, and patterns. Section V discusses the public disclosure of data from websites and software, as well as limitations and challenges. Section VI outlines planned future endeavors, including completing code samples in our dataset, engaging the community, training AI models and agents for integration into our VAITP tool, evaluating their performance, and exploring potential applications. Finally, Section VII concludes the document and summarizes the main contributions of our work.

II. RELATED WORK ON VULNERABILITY CLASSIFICATION

Problem classification has been demonstrated to aid in identifying clusters where systematic errors are likely to occur [19].

[1] A Python package is a collection of related modules.

[2] A Python library is a broader term that refers to a collection of modules and packages.

[3] CVE Details, CVE.org, and Snyk websites gather and organize information about standardized identifiers for publicly known security vulnerabilities. CVE Fixes is an SQLite database that gathers git commits related to vulnerability corrections in public GitHub repositories.

[4] AI-in-the-loop (AIiTL) integrates AI into human processes, with AI contributing insights or automation while humans oversee and validate outcomes.

[5] AI hallucinations refer to inaccurate or nonsensical outputs generated by AI models, often stemming from unexpected or erroneous patterns learned during training.

However, classification can be subjective and may not always yield consensus [20]. In this section, we delve into defects, faults, and vulnerability classification schemes and tools, how these influence our work, and how our work is relevant in the presented context.

In software security, vulnerability classification is crucial for understanding and addressing potential threats. Established practices like Orthogonal Defect Classification (ODC) have been used to systematically categorize defects [5]. ODC, introduced by Chillarege et al., uses semantic information to elucidate cause-effect relationships in the development process [21]. This approach has had a significant impact on subsequent works, including research on software fault emulation. A field data study conducted by Madeira et al. provides practical insight into software fault emulation and introduces Code Defects Classification (CDC) [21], [22], further enhancing defect classification understanding. Both works have influenced our taxonomy with minor adaptations, elaborated in Section III-B.

Landwehr et al. presented a taxonomy for computer program security flaws and documented 50 actual security flaws from open literature [23]. Inspired by their work, we systematically documented, classified, and reviewed 1026 Python-related vulnerabilities.

Avizienis et al. offer a comprehensive framework for dependable and secure computing systems. This framework covers fault tolerance, reliability, availability, safety, and security. It's applicable to Python vulnerabilities like injection attacks, insecure deserialization, or inadequate input validation.

Margarido et al. [20] explored defect classification tailored to specific software contexts. Their work highlights the importance of adapting defect classification to the nuances of different software artifacts, which is particularly relevant for Python vulnerabilities.

Li et al. proposed a vulnerability classification scheme with four categories: Bohr-Vulnerability (BOV), Non-Aging-Related Mandel Vulnerability (NMV), Aging-Related Vulnerability (ARV), and Unknown Vulnerability (UNK). Despite its unique perspective, this categorization was not included in our taxonomy, as it did not offer valuable insights for injecting or exploiting Python-specific vulnerabilities.

Wei et al. conducted a comprehensive study on security vulnerabilities, proposing a taxonomy based on root cause, consequence, and location. Analyzing 1076 bug reports from major projects in the National Vulnerability Database [24]. We incorporated parts of this classification relevant to Python vulnerabilities, like *Input Validation Errors*, which are also found in the Code Defects Classification (CDC) [21], [22].

Jiang et al. introduced a taxonomy for CPython[6] vulnerabilities, categorizing them into broad categories [25]. However, this taxonomy lacks the granularity needed for a high-level language like Python, limiting its usefulness for developers, researchers, and security teams.

While some taxonomies target low-level languages, they are not suitable for high-level language vulnerabilities like Python [2]. Our taxonomy separates root causes from potential impacts as distinct fields. For example, we consider *Buffer overflow* not as the root cause, but rather as a symptom linked to improper input data checking and validation, manifesting as buffer overflow.

There are also several tools available that use their own classification scheme, often not interchangeable with each other. For example, Bandit's detection of a call to *subprocess* with the shell parameter set to *True* is categorized with a *test ID* of *B602*. It also includes parameters such as the test name (*subprocess_popen_with_shell_equals_true*), severity (*HIGH*), confidence level (*HIGH*), CWE (*CWE-78*), and a link to documentation providing a detailed description of the vulnerability [13]. On the other hand, Semgrep's detection of the same function call has a *rule ID* of *ruleid:dangerous-subprocess-use-tainted-env-args*, along with a description of the vulnerability, test code, and references [14]. These differences in classification and reporting can pose challenges for users trying to compare or integrate results from different tools. The unified approach that we use in our dataset will be able to solve this problem, making it easier to see that the data is the same.

The dynamic landscape of software security demands continuous advancements in vulnerability classification. Python faces unique challenges from vulnerabilities within its code, libraries, or packages, prompting efforts to refine defect classification specifically for Python vulnerabilities. Reviewing the literature shows that merging traditional defect classification taxonomies with domain-specific adaptations enhances vulnerability analysis, improving our understanding of Python vulnerabilities.

In this article, we analyze 1026 Python vulnerabilities, offering a specific perspective on security flaws within the Python programming language. Our classification scheme covers a wide range of attributes: these include ODCs, CDC, short and long descriptions of vulnerabilities, CVE and CVE links, impact risk score, publication date, CWE and CWE links, and OWASP Top 10 classifications. Besides the mentioned categorizations, which were scraped, we also classified all vulnerabilities in our dataset into accessibility scope (local/remote/both), category, subcategory, possible impact, solution, and manually reviewed patched and vulnerable code samples. Additionally, it offers information on whether the vulnerability originates from Python itself or a third-party package. Furthermore, it details whether existing detection tools list the vulnerability in their documentation and if they detect the vulnerable code sample. Finally, we have included links to the sources we identified throughout our research. This comprehensive set of attributes offers a detailed and thorough perspective on the landscape of Python vulnerabilities, enabling statistical analysis to correlate the relationship

[6]CPython is the reference implementation of the Python programming language. It is written in C and provides the standard for the Python language specification.

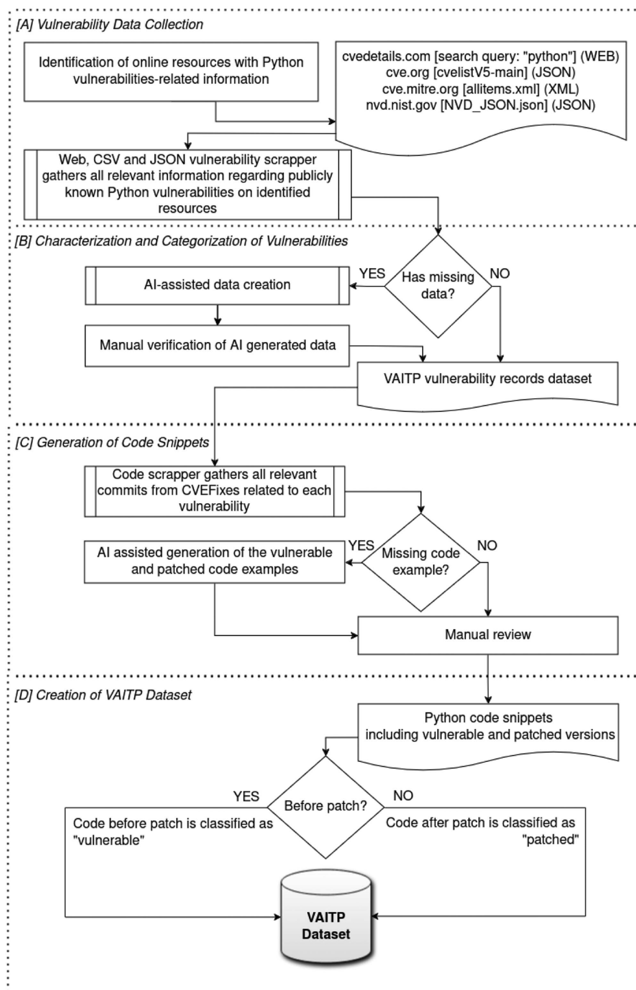

FIGURE 1. Methodology overview.

between attributes (e.g., how vulnerability clusters of a particular ODC category tend to be publicly disclosed over time). Moreover, all code samples have been rigorously tested to ensure accurate execution.

III. METHODOLOGY

This section outlines our approach, detailing the data sources, data scraping intricacies, classification criteria, and the process for both scraping and AI-based generation of code samples.

An overview of our methodology is presented in Fig. 1.

A. VULNERABILITY DATA COLLECTION

Our primary data source for Python vulnerabilities is the well-known CVEDetails.com, a website that aggregates information about Common Vulnerabilities and Exposures (CVEs), standardized identifiers for publicly known security vulnerabilities. We developed an HTML web scraper to gather Python-related vulnerabilities from CVEDetails.com by parsing all result pages obtained from querying the source with the keyword *Python* (conducted on January 11, 2024). The web scraper identified a total of 965 vulnerabilities, which comprise 94% of our dataset.

We also developed a JSON vulnerability scraper to extract all Python-related vulnerability information from the 236,637 JSON records existing in the cve.org dataset. This effort resulted in a 6% increase over the CVEDetails data gathered, identifying 61 more vulnerabilities for a total of 1,026 Python-related vulnerabilities. Additionally, we created two more scrapers: one for the cve.mitre.org dataset in XML format and another for the nvd.nist.gov dataset in JSON format. From these datasets, we extracted 844 and 749 Python-related vulnerabilities, respectively. However, neither of these datasets contributed to an increase in the number of the already identified vulnerabilities. The lack of new findings from these last two sources indicates a saturation in the gathered data, suggesting that our research methodology was comprehensive and effectively captured the breadth of Python-related vulnerabilities in the available datasets.

To unify all scrapers, we developed a script that sequentially executes all of them, optionally keeping a log of each execution. All the mentioned web scrapers and the vulnerability monitor can be found in the project's GitHub repository [26].[7]

The information collected includes CVE identifiers, descriptions, scores, and publication dates. Less than 10% of the collected data had missing elements that needed to be filled in. To address this, as well as supplement additional categorizations not available in the original source, we employed an AI-in-the-loop (AIiTL) approach to review each output, providing the AI models (e.g., OpenAI's GPT-3.5/4) with information about a vulnerability and using them to better understand the nature and possible classifications that could be applicable to each vulnerability.

B. CHARACTERIZATION AND CATEGORIZATION OF VULNERABILITIES

We started the characterization and categorization processes by selecting established taxonomies to classify defects, faults, and vulnerabilities. The ODC by Chillarege [5] and the CDC by Duraes and Madeira [21], [22] were selected as being the most adequate taxonomies. Additionally, we drew insights from various online resources, such as CVEdetails.com [8], CWE [6], and OWASP [7].

ODC and CDC offer a systematic approach to understanding software defects and faults, categorizing them according to various semantic attributes. CWE provides a standardized language for describing security vulnerabilities in software, facilitating effective communication and analysis among diverse stakeholders. OWASP's Top 10 offers a comprehensive list of the most critical web application security risks, guiding developers and security professionals in prioritizing efforts to mitigate these risks.

[7] The community can contribute to the GitHub repository with new vulnerability code examples. These examples are automatically loaded and presented on VAITP's website.

In addition to existing taxonomies, we developed our own taxonomy, comprehensively enumerated and explained next in this section.

Upon compiling the list of vulnerabilities, which primarily included CVE identifiers, vulnerability descriptions, publication dates, and risk scores, we proceeded to systematically characterize them based on various criteria sourced from online resources, including ODC, CDC, CWE, and OWASP. Additionally, we categorized vulnerabilities by their accessibility scope (local or remote). Short descriptions of each vulnerability were created, and source reference links were gathered to provide comprehensive information.

We initiated the taxonomy proposal process by identifying taxonomies relevant to Python vulnerabilities, as detailed in Section II. Each identified vulnerability was then assessed to determine the most appropriate category based on its description. The resulting taxonomy comprises 10 overarching categories (such as *Cryptographic* for vulnerabilities pertaining to encryption issues) and 41 subcategories (which provide more specific classifications within each broader category, such as *Improper SSL/TLS Certificate Validation* for vulnerabilities associated with inadequate validation of cryptographic certificates; please refer to Table 3).

To finalize our dataset, we adopted an AIiTL methodology to verify each vulnerability's categorization and generate vulnerable and patched code samples. This approach involved interaction with AI models using prompt engineering and rigorous manual scrutiny, with a particular emphasis on addressing instances of model hallucinations. To mitigate this issues, rigorous cross-validation with verified datasets was conducted, and human oversight was incorporated to verify outputs.

To analyze and ensure data consistency in the dataset (e.g., no missing records, no duplicates), we utilized Data Analyst GPT4, prompting it accordingly with our attached data set, a technique proven to achieve performance comparable to that of a senior data analyst [27].

Fig. 2 depicts the methodology utilized to establish classifications to categorize the identified vulnerabilities. The website provides a platform for the community to review these classifications and suggest modifications. Contributions submitted through the website will contribute to enhancing the accuracy of the proposed Python taxonomy and ensuring the correct classification of vulnerabilities within the dataset, keeping it up to date. Further elaboration on the features of the website is provided in Section VI.

To systematically categorize and analyze the identified Python vulnerabilities, a comprehensive classification scheme was employed. This scheme draws upon established taxonomies and incorporates domain-specific adaptations tailored to the intricacies of Python vulnerabilities. The ODC framework served as a foundation for understanding the types and nature of defects. Within ODC, *Orthogonal Defect Type* refers to the categorization of defects based on their manifestation, such as functional deficiencies, interface issues, or algorithmic flaws. *Code Defect Classification*

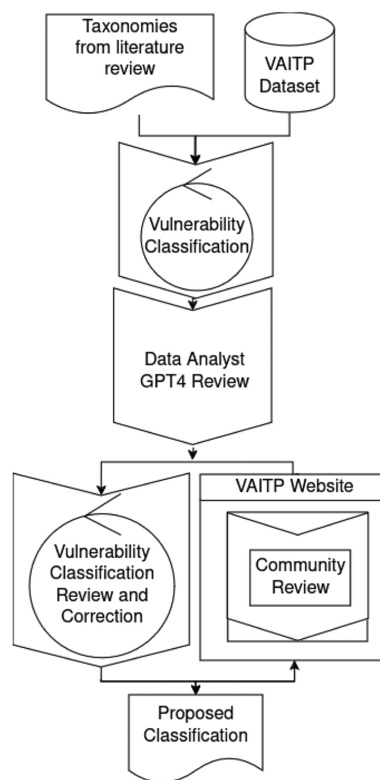

FIGURE 2. **Classification process.**

(CDC) classifies whether a defect stems from missing, incorrect, or extraneous functionality or logic. Complementing, the CWE and OWASP Top 10 taxonomies provided standardized languages for describing security weaknesses and prioritizing web application risks, respectively. Additionally, a custom taxonomy specific to Python vulnerabilities was developed, encompassing categories such as input validation, cryptographic flaws, and memory corruption, each with granular subcategories to capture the nuances of these security issues. Our dataset includes the following taxonomy components and their attributes for classifying vulnerabilities:

1) *Orthogonal Defect Type:* According to an adaptation of ODC [5], each vulnerability can be categorized into different types: *Function*, *Interface*, *Checking*, *Assignment*, *Timing/Serialization*, *Build/Packet/Merge*, or *Algorithm*. Refer to Table 2 for details.
2) *Code Defect Classification:* According to the CDC classification, the nature of each vulnerability can be classified as *Missing*, *Incorrect*, or *Extraneous* [21], [22]. Refer to Table 2 for details.
3) *Vulnerability Description:*
   - *Short Description:* A concise one-line reference to the vulnerability.
   - *Long Description:* A detailed explanation of each vulnerability, including manually reviewed information and relevant research findings.
4) *Common Vulnerabilities and Exposures (CVE):*

TABLE 1. Summary of Data Origin

Dataset Component	Scraped	AIiTL
ODC	0	1026
CDC	0	1026
Score	988	38
CWE	621	405
OWASP	621	405
Accessibility Scope	0	1026
Category	0	1026
Subcategory	0	1026
Impact	0	1026
Solution	0	1026
Patched code	171	855
Vulnerable code	171	855

TABLE 2. Code Defects and Fault Nature

ODC	Total	CDC	Total	Example
Function	737	Missing	9	A new function or functionality in an existing function is missing
		Incorrect	704	Part of the function code structure needs to be altered
		Extraneous	24	Functionality that is actually not needed
Interface	40	Missing	2	A parameter is missing in a function call
		Incorrect	37	Incorrect information was passed to a function call
		Extraneous	1	Surplus data was passed to a function call
Checking	106	Missing	61	Conditional logic is missing
		Incorrect	44	Incorrect logic used
		Extraneous	1	Superfluous logic should not be present
Assignment	11	Missing	1	A variable was not assigned a value or not initialized
		Incorrect	10	An incorrect value was assigned to a variable
		Extraneous	1	A variable should not have been assigned
Timing/ Serialization	50	Timing Issues	29	Thread issues or race conditions
		Serialization Issues	20	Incorrect serialization operations
		Extraneous Serialization	1	Superfluous serialization
Build/ Package/ Merge	27	Building Issues	2	Undefined reference to function
		Packaging Issues	25	Conflict of dependency versions
		Merging Conflicts	0	Two branches of a git changed the same line of code
Algorithm	55	Missing	2	Part of the algorithm is missing
		Incorrect	52	Algorithm is not correctly coded
		Extraneous	1	Algorithm has superfluous code

- *CVE Identifier:* Unique reference number for a vulnerability.
- *CVE Link:* Link to the CVE details page.

5) *Risk Score:* Severity of the vulnerability's impact, ranging from 0 (lowest) to 10 (highest), based on the Common Vulnerability Scoring System (CVSS) [28].
6) *Publication Date:* Date the vulnerability was first published on any of the sources.
7) *OWASP Top 10 Categorization:* Classification of the vulnerability according to OWASP Top 10 (2021 list) [7].
8) *Vulnerability Accessibility Scope:* Classification as *Local Application Vulnerability* indicating a vulnerability arising from vulnerabilities that can only be exploited having physical access to the system; and *Remote Application Vulnerability*, which arises from vulnerabilities that can be exploited remotely without having physical access to the system.
9) *Category and Subcategory:* Our defined categories for Python vulnerabilities, including 1) *Input Validation and Sensitization*, 2) *Authentication, Authorization, and Session management*, 3) *Cryptographic*, 4) *Design Defects*, 5) *Configuration Issues*, 6) *Memory Corruption*, 7) *Information Leakage*, 8) *Race Condition*, 9) *Resource Management*, and 10) *Numeric Errors*. Each Category includes several subcategories (see Table 3).
10) *Impact and Threat:* Possible consequences of the vulnerability, such as information disclosure or arbitrary code execution.
11) *Solution:* Steps to correct the vulnerability.
12) *Patched and Vulnerable Code Samples:* Python code samples of patched and vulnerable versions of the code obtained from the CVEFixes database or generated by AI models.
13) *Python / Library:* Field that indicates whether the vulnerability affects Python itself or a specific library or package (e.g., *Django*).
14) *Detection Tools Listing:* Indicates whether tested vulnerability detection tools list the vulnerability in their documentation.
15) *Vulnerable Code Sample Detection:* Indicates whether tested vulnerability detection tools detect the vulnerable code sample.
16) *References:* Links of the sources obtained during research.

This classification and these attributes play an important role in systematically categorizing and analyzing Python vulnerabilities and provide a structured approach to understanding their nature, impact, and mitigation strategies. By leveraging established taxonomies, along with our own tailored classifications, we provide a comprehensive and nuanced perspective on the vulnerabilities. Attributes such as risk score, publication date, and proof of concept facilitate accurate assessment and prioritization of vulnerabilities, enabling effective vulnerability management and mitigation efforts. Additionally, the inclusion of patched and vulnerable code samples enhances the practical relevance of the dataset, empowering developers and security professionals to better understand and address Python vulnerabilities.

An overview of the data that could be automatically gathered through scraping and the data that needed to be generated afterward using an AI-in-the-Loop (AIiTL) approach is depicted in Table 1. The data that could not be scraped from the sources includes some ODC and CDC classifications, several code snippets, determining the accessibility scopes, categories, subcategories, impacts, and possible solutions (e.g., updating to a specific Python or library version).

TABLE 3. Categories and Subcategories of Vulnerabilities

Category	Total	Subcategory	Total	Description
Input Validation and Sanitization	459	Command Injection	105	Injection of arbitrary commands into user input.
		SQL Injection	4	Injection of SQL from user input in a database management system.
		Insecure Direct Object References (IDOR)	31	Unauthorized access to objects by manipulating references.
		Path Traversal	52	Unauthorized access to files and directories by manipulating file paths.
		Insecure Parsing or Deserialization	267	Security issues during the deserialization or parsing of data.
Authentication, Authorization, and Session Management	89	Weak Password Policies	1	Inadequate password security measures.
		Insecure Authentication Mechanisms	31	Flaws in the authentication process.
		Session Management Issues	2	Vulnerabilities related to session handling and management.
		Privilege Escalation	55	Unauthorized elevation of user privileges.
Cryptographic	84	Unencrypted communication	9	Plain-text communication allows sniffing of sensitive data.
		Weak encryption algorithms	2	Weak encryption of sensitive data.
		Inadequate random number generation	8	Generation of inadequate random numbers.
		Improper SSL/TLS Certificate Validation	37	Improper validation of SSL/TLS Certificates.
		Cryptographic Implementation Error	28	Vulnerabilities related to mistakes or flaws in cryptographic algorithms, methods, or libraries.
Design Defects	73	Inadequate Error Handling	11	Insufficient handling of unexpected errors or exceptions, potentially exposing sensitive information or causing system instability.
		Vulnerable and Outdated Components	38	Outdated and deprecated components that introduce a known vulnerability.
		Poorly Designed Access Controls	5	Flaws in how the system manages user privileges and permissions, leading to unauthorized access.
		Security Misconfigurations	19	Insecure configuration leading to vulnerabilities.
Configuration Issues	68	Cross-Site Scripting (XSS)	28	Injecting malicious code into web apps to compromise user data or actions.
		Cross-Site Request Forgery (CSRF)	8	Unauthorized execution of actions through forged requests.
		Remote File Inclusion (RFI)	3	Inclusion of remote files in web applications.
		Local File Inclusion (LFI)	1	Inclusion of local files in web applications.
		Open Redirects	15	Improper handling of redirection URLs.
		Server-Side Request Forgery (SSRF)	9	Tricking the server to make unauthorized requests.
		Dynamic Link Library (DLL) Loading Issues	4	Improper handling of dynamic libraries, potentially allowing malicious DLLs to be loaded and executed.
Memory Corruption	98	Buffer Overflows	52	Occurs when a program writes more data to a buffer than it can hold, potentially overwriting adjacent memory.
		Out-of-Bound Accesses	32	Involves accessing memory locations outside the allocated boundaries, often leading to unintended consequences.
		Use-After-Free Errors	14	Refers to using memory after it has been deallocated, potentially causing unpredictable behavior or vulnerabilities.
Information Leakage	44	Information Disclosure	18	Accidental exposure of sensitive information related to a system.
		Insecure Handling of Sensitive Data	26	Mishandling and exposure of sensitive information related to a user.
Race Conditions	14	Time-of-Check to Time-of-Use	2	Situations where the state of a resource changes between the time it is checked and the time it is used, leading to unexpected behaviors.
		Data Race Conditions in Threads	1	Occurs when multiple threads or processes concurrently access and modify shared data, potentially resulting in unpredictable outcomes.
		Race Conditions in File Operations	11	Race conditions that specifically affect file operations, which may result in security vulnerabilities when handling files.
Resource Management	65	File Handle Leaks	3	Failure to release file handles after use, potentially leading to resource exhaustion or security vulnerabilities.
		Socket Handle Leaks	1	Neglecting to close network socket handles, which can result in resource depletion or potential security issues.
		Memory Leaks	5	Failing to deallocate memory properly, causing the program to consume excessive memory resources.
		Resource Exhaustion	56	Depleting system resources, such as CPU, memory, or network connections, due to poor resource management, potentially leading to system instability or denial of service.
Numeric Errors	32	Integer Overflows	25	Occur when integer variables exceed their maximum values, often leading to unexpected or insecure behavior.
		Rounding Errors	2	Result from imprecise rounding of numerical values, potentially causing discrepancies in calculations.
		Floating-Point Precision Issues	1	Stem from the finite precision of floating-point numbers, potentially causing inaccuracies in mathematical operations.
		Arithmetic Errors	4	Involve mistakes in numerical calculations, which can lead to unintended results or vulnerabilities in software.

C. GENERATION OF CODE SNIPPETS

We developed a code scraper for CVEFixes [12], collecting 171 pairs of vulnerable and patched code samples, while the remaining 855 examples were generated using GPT models. Leveraging the contextual awareness and proficiency of GPT models in understanding programming languages and security concepts [29], we interacted with the AI models to classify vulnerabilities and generate code samples. Our process involved providing the model with CVE ID and vulnerability descriptions, reviewing the output for accuracy, and recording the reviewed values for each vulnerability. This interaction demonstrated the efficacy of prompt engineering with GPT models in vulnerability classification and code generation. However, recent interactions with GPT-4 revealed inconsistencies in generating vulnerable code, highlighting ongoing challenges in optimizing AI models for specific tasks, particularly those involving ethically sensitive content. Further refinement is needed to address such inconsistencies.

D. CREATION OF VAITP DATASET

Each collected Python code snippet was subsequently reviewed and categorized as either belonging to the *vulnerable* or *patched* vulnerability cluster. These were then pushed to VAITP's GitHub repository [26].

IV. PYTHON VULNERABILITY DATA ANALYSIS

This section analyzes our dataset of vulnerabilities, focusing on the number, distribution, and trends.

Our dataset comprises 1026 Python vulnerabilities, classified according to the attributes described in Section III-B.

Table 2 illustrates the distribution of vulnerabilities based on ODC and CDC Defect Nature. The predominant category of vulnerabilities pertains to *Incorrect Functionality*, suggesting that functions within the underlying code exhibit incorrect behavior, needing modification to align with expected functionality.

Table 3 displays the number of vulnerabilities for each category and subcategory in our dataset. *Input Validation and Sanitization* is the most common category, with 446 vulnerabilities (43%), primarily attributed to *Insecure Parsing or Deserialization*. The analysis also reveals a significant number of vulnerabilities in the *Memory Corruption* and *Authentication, Authorization, and Session Management* categories. Our analysis indicates an increase in publicly disclosed Python vulnerabilities in recent years, peaking in 2022. Although the scraping process was successful, a small amount of data (i.e., less than 5%) was missing from the impact score, the CWE, and the OWASP fields. The origin of the missing data from the web scraping process is detailed in Table 1. Most of the missing data was generated using AI with an AIiTL approach based on the vulnerability descriptions.

We conducted an analysis of the distribution of vulnerabilities over time, examining correlations between categories and their respective publication dates. Fig. 3 illustrates the number of published CVEs related to Python per publication date.

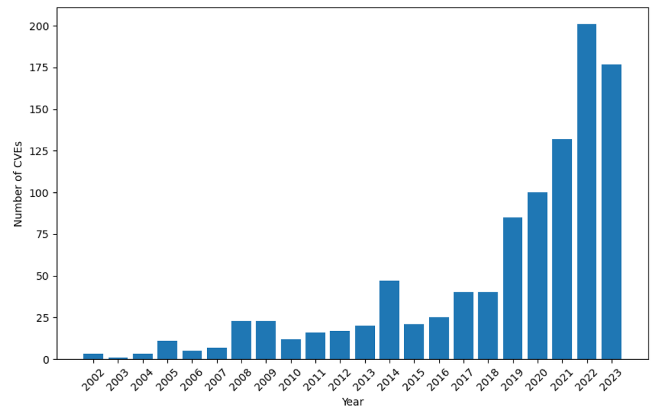

FIGURE 3. Number of python-related CVE vulnerabilities published per year.

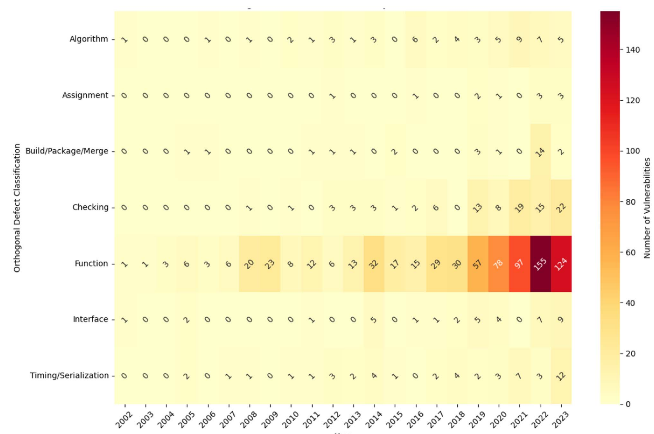

FIGURE 4. Heat map of the relationship between the ODC and the vulnerabilities' publication year.

The graph indicates a noticeable increase in vulnerabilities over time, with a peak observed in 2022, as expected. However, there is a slight decrease in the number of vulnerabilities reported in 2023. This trend suggests a growing prevalence of Python vulnerabilities, possibly influenced by the increasing popularity and usage of Python, as well as heightened attention from the security community.

In Fig. 4, we observe the relationship between the ODC classification and their corresponding publication dates. The data reveals that a substantial number of defects reported during this period are associated with *function* issues, with a notable peak observed in 2022. This trend indicates a growing prevalence of functional defects in Python code over time.

In Fig. 5, we examine the relationship between the Code Defect Classification of vulnerabilities and their respective publication dates. The data highlight a consistent trend, corroborating the observations from Fig. 4, where the most prevalent type of nature for Python vulnerabilities is attributed to *Incorrect Functionality*. This trend underscores the importance of functional defects in Python code and suggests a persistent challenge to ensure the correctness and reliability of Python applications over time.

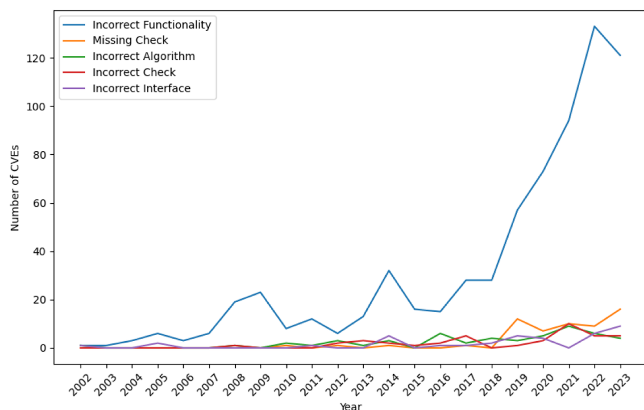

FIGURE 5. Relation between Nature and Publication year.

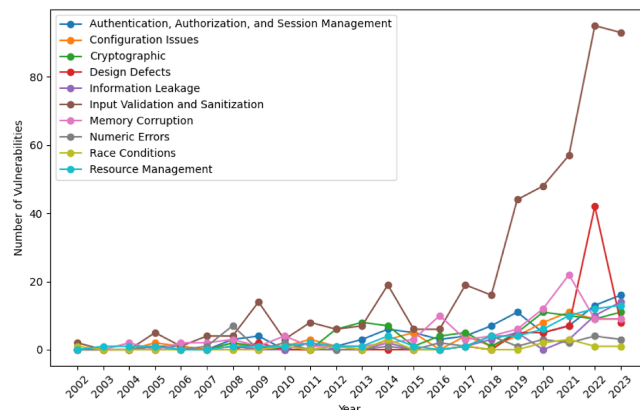

FIGURE 7. Relation between category and publication year.

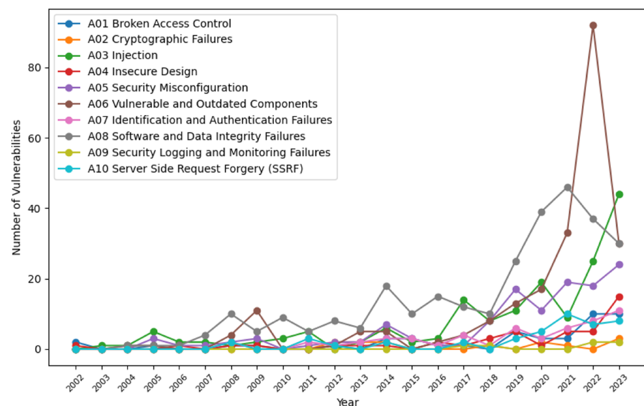

FIGURE 6. Relation between OWASP top 10 and publication year.

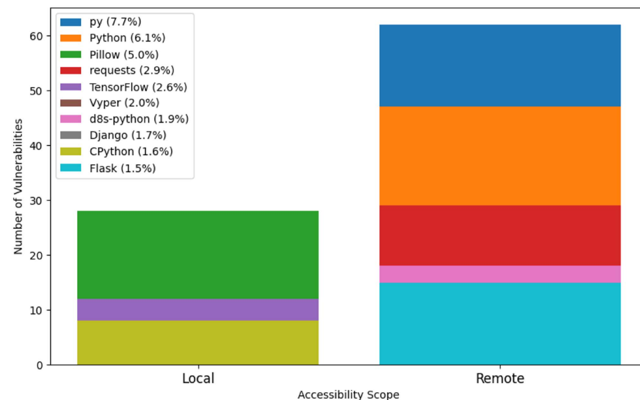

FIGURE 8. Relation between the accessibility scope of vulnerabilities and the most common libraries, packages, and Python versions.

Fig. 6 depicts the relationship between the OWASP Top 10 classification of Python vulnerabilities and their respective publication dates. This visualization provides insights into the distribution of vulnerabilities across OWASP's critical security risks over time.

By examining this relationship, we gain a better understanding of how Python vulnerabilities align with OWASP's prioritized list of security concerns and how these trends evolve over time.

The data indicates that the most prevalent OWASP classification types for Python vulnerabilities in 2023 were:

1) A03 Injection
2) A08 Software and Data Integrity Failures
3) A06 Vulnerable and Outdated Components

The prevalence of vulnerabilities related to *A03 Injection* in 2023 contrasts with the preceding year, where there was a spike in *A06 Vulnerable and Outdated Components*-related vulnerabilities.

Fig. 7 illustrates the relationship between the top ten categories of Python vulnerabilities and their publication dates. The data indicates a consistent trend, with the most prevalent category being *Input Validation and Sanitization* errors. This finding underscores the importance of addressing issues related to user input and external data validation and sanitization in Python code to improve security.

TABLE 4. Descriptive Statistics for the Dataset

Property	Mean	Median	Mode	Std	Min	Max
Score	6.59	6.5	7.5	1.76	1.2	10

Analyzing the frequency of the top five subcategories of Python vulnerabilities over time reveals that the most prevalent subcategory is *Insecure Parsing or Deserialization*, followed by *Command Injection*. This trend corroborates the data observed in Fig. 4, further proving the need to secure functions that deal with user input.

Fig. 8 demonstrates the relationship between the accessibility scope of vulnerabilities (local or remote) and the most common libraries, packages, and Python versions.

Table 4 summarizes the descriptive statistics for the vulnerability scores in our dataset, with a scale range from 1 to 10, representing low to high-risk scores. The average score for Python vulnerabilities is 6.59, indicating a medium-high level of risk. The score that occurs most frequently is 7.5, which also suggests a medium-high level of risk. The standard deviation of 1.76 reflects moderate variability in vulnerability scores around the mean. The highest score recorded is 10.00, representing a critical level of risk, while the lowest score is 1.2, indicating a very low level of risk.

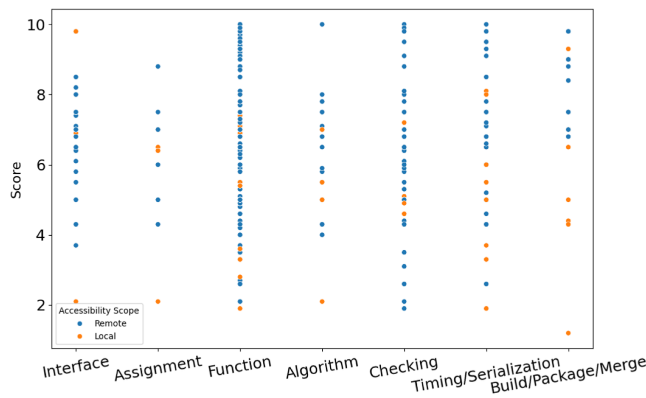

FIGURE 9. Scatter plot of the score vs. ODC.

Fig. 9 shows the relationship between vulnerability scores and the ODC classification. It is observable that most vulnerabilities with a high impact score are concentrated in the *Function* ODC category and have mainly a *Remote* Accessibility Scope (denoted by the blue dots). It is also observable that most *Local* vulnerabilities are rather related to building and packaging issues instead.

In summary, our analysis of Python vulnerabilities reveals several key findings: Over time, there has been a noticeable increase in reported vulnerabilities, with the majority being remotely exploitable and posing significant risks such as arbitrary code execution. Notably, the most prevalent types of vulnerabilities are related to input validation and sanitization, primarily attributed to insecure parsing or deserialization. Furthermore, our investigation highlights the importance of prompt updates, as only 16 out of 1026 vulnerabilities could not be remediated by updating the library, package, or Python to a higher version. This underscores the critical role of maintaining up-to-date systems to mitigate potential security risks effectively. However, cases such as CVE-2021-29921, where it was the updating of Python to 3.8.0 or 3.9.0 that would introduce the vulnerability, highlight the complex dichotomy between the necessity of keeping systems updated to prevent vulnerabilities and the risks associated with rapid release cycles and rolling updates, emphasizing the inherent risks of continuously updating systems without thorough testing, and underscoring the importance of stability and rigorously vetted updates in critical environments. These incidents serve as a reminder of the delicate balance between agility and security in software development and deployment, emphasizing the need for comprehensive testing, validation, and risk assessment protocols in update management strategies.

V. PUBLIC DATA DISCLOSURE

A. WEBSITE AND SOFTWARE TOOL

In addition to our dataset, we have developed a website to showcase our research and facilitate user interaction with our data and AI models. Here is an overview of the features available:

- Statistics about Python vulnerabilities, including their number and distribution by category, score, publication date, and impact, are presented with visualizations such as histograms and graphs to illustrate trends and patterns. The website offers insights on common and severe vulnerabilities, along with best practices and recommendations for Python security. Due to space constraints, we cannot include all analyses in this article. For a more comprehensive examination, refer to the supplementary material on our website.
- Access to dataset content, providing detailed information and properties of each vulnerability, such as code defect classification, nature, descriptions, etc. Users can browse, search, filter, and download data based on their research objectives.
- Community contribution feature that allows users to contribute to the dataset and taxonomy. Contributors are rewarded with our own token (see Section VI).
- General information about our AI models and software (VAITP).

Our website and software tool serve as valuable resources for the community, offering a comprehensive and interactive platform to learn about and explore Python vulnerabilities, as well as to test and enhance the security of Python scripts.

B. LIMITATIONS AND CHALLENGES

Not all vulnerabilities were included in the dataset, either due to the actual root cause of the vulnerability not lying in Python code or not being related to code at all. For example, CVE-2023–5625 and CVE-2021–24105 were excluded because they do not directly pertain to code-related vulnerabilities. The former involves a modification of the patch application strategy to the *python-eventlet* by the Red Hat team, while the latter concerns how attackers may exploit vulnerabilities in the package manager configuration to introduce malicious packages into the repositories. Such compromises could lead to remote code execution during the development, build, and release processes.

We have identified vulnerabilities in our dataset that include identical examples of vulnerable and patched code. For instance, CVE-2022-28470, affecting the *marcador* package, includes an incorrect reference to the *requests* package in its setup script from versions 0.1 to 0.13. However, the actual Python code a programmer can write using this package remains unchanged between affected and nonaffected versions. Similarly, several vulnerabilities associated with the *pillow* package demonstrate this pattern, where the vulnerability originates in the package's C code, leaving the Python code unaffected.

It is worth noting that relying solely on publicly disclosed vulnerabilities has limitations that affect the generalization of findings. Many vulnerabilities remain undiscovered or undisclosed, which can result in biased analysis and limited applicability across different environments. Although publicly

disclosed vulnerabilities provide valuable information, they do not provide a complete picture, thereby limiting the scope and applicability of research conclusions.

VI. FUTURE ENDEAVORS

Current research lays the foundation for further advancements in the field of Python vulnerability analysis and mitigation. Several avenues for future work have been identified, each aimed at enhancing the comprehensiveness and effectiveness of research results.

A. COMPLETION OF CODE SAMPLE COLLECTION

The ongoing effort to collect code samples associated with vulnerabilities is crucial for enriching the dataset. Completing this collection will provide a more exhaustive and representative set of vulnerabilities, contributing to a comprehensive understanding of Python security issues. Furthermore, we will be using Bandit, Semgrep, and SonarSource as static analysis tools to test the code samples for vulnerabilities. We will use the following steps to test the code samples:

1) Check tool documentation for a given vulnerability and record if it is listed.
2) Run Bandit, Semgrep, and SonarSource on the vulnerable code sample and record their results with respect to the number of issues detected.
3) Compare the results with the expected results.

B. COMMUNITY INVOLVEMENT ON THE VAITP WEBSITE

The VAITP website serves as a user-friendly platform for interacting with the dataset, exploring vulnerabilities, and gaining insights into Python security issues. We aim to empower the security community, researchers, and practitioners to access, analyze, and contribute to the wealth of information provided. Additionally, community members can contribute to data correction and refinement within the dataset and are rewarded with VAITP Tokens for their valuable contributions.

C. VAITP TOKEN

The VAITP Token operates on Ethereum, recording transactions based on user actions on the VAITP website. Our upcoming article will delve into the token's architecture, functionality, and implications, highlighting blockchain's role in driving user engagement and collaboration within the cybersecurity community.

D. AUTOMATION OF VULNERABILITY IDENTIFICATION, CLASSIFICATION AND CODE SCRAPING/GENERATION

Automating vulnerability identification accelerates research and enhances efficiency. We aim to develop Python scripts to automatically discover and incorporate new vulnerabilities into our dataset, engaging the security community with each disclosure.

AI agents will automate code example generation for newly disclosed vulnerabilities, ensuring timely availability of comprehensive data for analysis and mitigation. Our dataset will mirror recent trends and advancements in Python vulnerabilities, augmenting its relevance and comprehensiveness over time. The evolution of AI models remains key to our future endeavors. Continuous refinement of AI models for vulnerability classification, code generation, injection, and exploitation will enhance their accuracy and adaptability. Our previous research demonstrates the high accuracy (>95%) of our AI-based models in identifying potential locations for vulnerability injection [4], underscoring their potential impact.

E. VAITP DEVELOPMENT

The Vulnerability Attack and Injection Tool for Python (VAITP) represents a practical solution for vulnerability testing, empowering teams to scan, inject, and exploit Python vulnerabilities effectively. Completing the development of VAITP and ensuring its user-friendliness will facilitate thorough analyses and exploitability assessments on Python applications by security practitioners.

These future work directions collectively aim to fortify the research outcomes, contributing to the proactive and dynamic field of cybersecurity. Embracing automation, enhancing AI capabilities, and providing accessible tools will play a pivotal role in advancing Python application security and fostering collaborative efforts within the community.

VII. CONCLUSION

In this ongoing research, we have conducted a comprehensive analysis of Python vulnerabilities, including data collection and classification, with the aim of exploring potential injection and exploitation scenarios. Our methodology involved the gathering of Python vulnerabilities from the most important web repositories and applications, the use of web and code scraping, and an AIiTL approach to classify, analyze, and generate code for Python vulnerabilities. The resulting dataset comprises 1026 identified vulnerabilities, enriched with code samples and statistical analysis results.

Vulnerabilities were systematically classified according to various mentioned criteria, including Orthogonal Defect Classification (ODC), Common Weakness Enumeration (CWE), and OWASP Top 10. Additionally, vulnerable and patched code samples were meticulously collected and generated, accompanied by relevant payloads to facilitate practical analysis. To gain insights into the dataset, descriptive statistics were calculated, including measures such as mean, median, mode, standard deviation, minimum, and maximum values for vulnerability scores. The analysis revealed a medium-high level of risk, with an average score of 6.59 on a scale from 1 (low) to 10 (high). Furthermore, exploratory data analysis was performed using visualizations and plots derived from the dataset. These analyses provided valuable insights into the landscape of Python vulnerabilities, notably highlighting the increasing prevalence of remotely exploitable vulnerabilities in recent years. This trend underscores the growing importance of addressing remote attack vectors and fortifying Python applications against potential threats originating from external sources.

The analysis of our data revealed notable trends and patterns in Python vulnerabilities. Following ODC and CDC, *Input Validation and Sanitization* emerged as the most common category, with a significant focus on *Insecure Parsing or Deserialization* from user input fields and external sources. These findings can be beneficial for developers, allowing them to focus their vulnerability prevention efforts on the most sensitive functions of their code.

The OWASP Top 10 classification indicated a prevalence of vulnerabilities related to *Vulnerable and Outdated Components*, *Software and Data Integrity Failures*, and *Injection*. Furthermore, there was an increasing trend in the number of vulnerabilities reported in recent years. This emphasizes the security community involvement and the importance of keeping systems up-to-date.

The website and software tool (VAITP) developed as part of this research (and that will be publicly released upon completion of the test phase) serve as valuable resources to the security community. The interactive platform enables users to explore the dataset, analyze vulnerabilities, and contribute to the evolving landscape of Python security. The VAITP tool facilitates the use of AI models for vulnerability analysis, injection, and exploitation, empowering security teams, researchers, and developers alike with the ability to test their systems' resilience against different vulnerabilities.

This research advances our understanding of Python vulnerabilities, providing a comprehensive and ever-evolving dataset, insightful analysis, and practical tools. The findings and resources generated contribute to ongoing efforts to enhance awareness of Python application security and foster a proactive approach to vulnerability management.

The following list summarizes the main contributions of the current article:

1) *Python Vulnerability Taxonomy:* The research categorized vulnerabilities into a taxonomy focused on Python vulnerabilities.
2) *Comprehensive Dataset:* The research provides a detailed dataset comprising 1026 identified Python vulnerabilities, meticulously classified, and enriched with code samples and payloads.
3) *Exploratory Data Analysis:* This includes:
   - *Trend Analysis:* Identifying notable trends and patterns in Python vulnerabilities, including the most common types.
   - *Descriptive Statistics:* Computing descriptive statistics to provide a quantitative assessment of risk levels.
   - *Temporal Analysis:* Examining the distribution of vulnerabilities over time, revealing an increasing trend with a peak in 2022.

Our findings can have significant implications for Python developers and users. By identifying critical categories and patterns of vulnerabilities, developers can prioritize their security efforts to address high-risk areas, enhancing the overall security posture of Python applications.

REFERENCES

[1] Tiobe, "Tiobe (2024) tiobe index," 2024. [Online]. Available: https://www.tiobe.com/tiobe-index/
[2] WhiteSource, "What are the most secure programming languages?" 2023. [Online]. Available: https://www.mend.io/most-secure-programming-languages/
[3] J. T. F. Transformation, "Guide for conducting risk assessments," 2012. [Online]. Available: https://doi.org/10.6028/nist.sp.800-30r1
[4] C. G. Frédéric Bogaerts, N. Ivaki, and J. Fonseca, "Using AI to inject vulnerabilities in python code," in *Proc. IEEE/IFIP 53rd Annu. Int. Conf. Dependable Syst. Netw. Workshops*, 2023, pp. 223–230.
[5] R. Chillarege et al., "Orthogonal defect classification - a concept for in-process measurements," *IEEE Trans. Softw. Eng.*, vol. 18, no. 11, pp. 943–956, Nov. 1992.
[6] T. M. Corporation, "Common weakness enumeration (CWE)," 2023. [Online]. Available: https://cwe.mitre.org
[7] OWASP, "Owasp top ten," 2023. [Online]. Available: https://owasp.org/www-project-top-ten/
[8] Cvedetails, "Python CVE security vulnerabilities, versions and detailed reports," 2023. [Online]. Available: https://www.cvedetails.com/
[9] Cve.org, "CVE security vulnerabilities," 2024. [Online]. Available: https://www.cve.org/
[10] Snyk, "Snyk - developer security - develop fast. stay secure. - snyk," 2023. [Online]. Available: https://snyk.io/
[11] NIST. "National vulnerability database (NVD)." 2019. Accessed: 2024. [Online]. Available: https://nvd.nist.gov/
[12] G. Bhandari, A. Naseer, and L. Moonen, "CVEfixes: Automated collection of vulnerabilities and their fixes from open-source software," in *Proc. 17th Int. Conf. Predictive Models Data Analytics Softw. Eng.*, Aug. 2021, pp. 30–39.
[13] Bandit, "Blacklist calls – bandit documentation," 2023. [Online]. Available: https://bandit.readthedocs.io/en/latest/blacklists/blacklist_calls.html
[14] Semgrep, "Semgrep," 2023. [Online]. Available: https://semgrep.dev/
[15] SonarSource, "Sonarsource rules," 2023. [Online]. Available: https://rules.sonarsource.com/python/type/Vulnerability/RSPEC-5334
[16] J. M. Maynez, S. Narayan, B. Bohnet, and R. McDonald, "On faithfulness and factuality in abstractive summarization," 2020, *arXiv:2005.00661*.
[17] S. Lin, J. Hilton, and O. Evans, "TruthfulQA: Measuring how models mimic human falsehoods," 2021, *arXiv:2109.07958*.
[18] F. Bogaerts et al., "Vaitp," 2023. [Online]. Available: https://netpack.pt/vaitp
[19] D. N. Card, "Learning from our mistakes with defect causal analysis," *IEEE Softw.*, vol. 15, no. 1, pp. 56–63, Jan./Feb. 1998.
[20] I. L. Margarido et al., "Classification of defect types in requirements specifications: Literature review, proposal and assessment," in *Proc. IEEE 6th Iberian Conf. Inf. Syst. Technol.*, 2011, pp. 1–6.
[21] J. A. Duraes and H. S. Madeira, "Emulation of software faults: A field data study and a practical approach," *IEEE Trans. Softw. Eng.*, vol. 32, no. 11, pp. 849–867, Nov. 2006.
[22] J. Duraes and H. Madeira, "Definition of software fault emulation operators: A field data study," in *Proc. IEEE/IFIP 43rd Annu. Int. Conf. Dependable Syst. Netw.*, 2003, pp. 105–114. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/DSN.2003.1209922
[23] C. E. Landwehr et al., "A taxonomy of computer program security flaws," *ACM Comput. Surv.*, vol. 26, no. 3, pp. 211–254, 1994. [Online]. Available: https://doi.org/10.1145/185403.185412
[24] Y. Yang et al., "A comprehensive study on security bug characteristics," *J. Softw., Evol. Process*, vol. 33, no. 10 , 2021, Art. no. e2376. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/smr.2376
[25] C. Jiang, B. Hua, W. Ouyang, Q. Fan, and Z. Pan, "PyGuard: Finding and understanding vulnerabilities in python virtual machines," in *Proc. IEEE 32nd Int. Symp. Softw. Rel. Eng.*, 2021, pp. 468–475.
[26] F. Bogaerts et al., "Vaitp," 2023. [Online]. Available: https://github.com/netpack/vaitp
[27] L. Cheng, X. Li, and L. Bing, "Is GPT-4 a good data analyst?" 2023, *arXiv:2305.15038*.
[28] A. Horváth, P. M. Erdósi, and F. Kiss, "The Common Vulnerability Scoring System (CVSS) generations – usefulness and deficiencies," 2016, pp. 137–153.
[29] T. B. Brown et al., "Language models are few-shot learners," *Adv. Neural Inf. Process. Syst.*, vol. 33, pp. 1877–1901, 2020.