Exploring Security Commits in Python

Shiyu Sun*, Shu Wang*, Xinda Wang*, Yunlong Xing*, Elisa Zhang†, Kun Sun*
*George Mason University, †Dougherty Valley High School
{ssun20, swang47, xwang44, yxing4, ksun3}@gmu.edu, elisaz.ca@gmail.com

***Abstract*—Python has become the most popular programming language as it is friendly to work with for beginners. However, a recent study has found that most security issues in Python have not been indexed by CVE and may only be fixed by "silent" security commits, which pose a threat to software security and hinder the security fixes to downstream software. It is critical to identify the hidden security commits; however, the existing datasets and methods are insufficient for security commit detection in Python, due to the limited data variety, non-comprehensive code semantics, and uninterpretable learned features. In this paper, we construct the first security commit dataset in Python, namely PySecDB, which consists of three subsets including a base dataset, a pilot dataset, and an augmented dataset. The base dataset contains the security commits associated with CVE records provided by MITRE. To increase the variety of security commits, we build the pilot dataset from GitHub by filtering keywords within the commit messages. Since not all commits provide commit messages, we further construct the augmented dataset by understanding the semantics of code changes. To build the augmented dataset, we propose a new graph representation named CommitCPG and a multi-attributed graph learning model named SCOPY to identify the security commit candidates through both sequential and structural code semantics. The evaluation shows our proposed algorithms can improve the data collection efficiency by up to 40 percentage points. After manual verification by three security experts, PySecDB consists of 1,258 security commits and 2,791 non-security commits. Furthermore, we conduct an extensive case study on PySecDB and discover four common security fix patterns that cover over 85% of security commits in Python, providing insight into secure software maintenance, vulnerability detection, and automated program repair.**

***Index Terms*—Security Commit, Python, Dataset Construction, Code Property Graph, Graph Learning, Vulnerability Fixes**

I. Introduction

According to the TIOBE index [1], Python overtakes Java and C as the most popular programming language as of April 2023. However, a recent study [2] reveals that among 749K security issues of 197K Python packages in PyPI [3], only 1,232 vulnerabilities are reported to CVE [4] and only 556 of them have public fixes. Thus, most security issues are not indexed and may only be resolved by "silent" fixes, without explicit log messages indicating the vulnerabilities.

The hidden security fixes pose a threat to the security and privacy of users, since attackers may exploit the undisclosed vulnerabilities to comprise the unpatched software systems. Particularly, Python is friendly to beginners; however, with limited security knowledge, learners may be unable to determine if an upstream commit is intended to address a vulnerability and hence neglect a critical security fix. The implicit security commits can impair software maintenance and evolution since the downstream developers may not be aware of the criticality of these security commits. Therefore, it is vital to identify the hidden security commits among all Python commits.

The existing datasets and methods are insufficient for security commit detection in Python. To identify security commits, researchers first build commit datasets [5], [6] and then either extract features manually from commit messages/-code changes [5], [7], [8] or learn features automatically via deep learning models, e.g., recurrent neural networks (RNN) [9], [10], Transformers [11], [12], and graph neural networks (GNN) [13], [14]. However, these solutions have three main constraints, namely, limited data variety, non-comprehensive code semantics, and uninterpretable learned features. First, existing works [5], [6], [15] only construct security commit datasets in C/C++ or Java, and there is no available Python dataset for security commit research. Second, the existing feature extraction methods cannot be directly applied to Python security commit detection. The manually extracted features [5], [7] are language-dependent and cannot be directly migrated to the Python language. Also, the existing deep learning features [9], [12], [13], [15] are incapable of integrating both the sequential and structural semantics of code since the RNN and Transformer-based models simply treat code as a natural language [16], [17] and the GNN-based models only focus on the code dependencies [13], [14]. Third, though manually extracted features are interpretable, feature extraction methods can only provide limited accuracy. In contrast, deep learning based methods may yield better accuracy, but their learned features are uninterpretable.

To tackle the above challenges, we construct the first security commit dataset in Python, named PySecDB,[1] by collecting the samples from CVE records and filtering the commits from GitHub. It consists of three subsets: a base dataset, a pilot dataset, and an augmented dataset. We first build a base dataset by collecting the security commits associated with CVE IDs [4]. Since the CVE records on Python programs are limited, we observe that only 46% of them provide the corresponding security commits and more security commits fall in the wild silently, without being indexed by CVE. To enrich security commits for covering more vulnerability types, we construct a pilot dataset by filtering GitHub commits. Since only 6-10% of GitHub commits are related to security fixes [5], filtering commit messages using relevant keywords can efficiently narrow down the list of security commit candidates.

[1]This dataset is released in https://github.com/SunLab-GMU/PySecDB.

The security keywords are automatically extracted from the commits in the base dataset by the Latent Dirichlet Allocation (LDA) method [18]. Then, three security experts manually verify the candidate commits and build the pilot dataset. Well-documented commit messages are required for the pilot dataset construction, but not all commits provide commit messages.

To include more diverse security commits that do not provide sufficient commit messages, we extend the base and pilot datasets with an augmented dataset by considering the semantics of code revisions. We develop a new commit graph representation named CommitCPG and a graph learning model named SCOPY to capture the semantics of code changes. Inspired by [19], our CommitCPG is constructed by merging the code property graphs of the previous and current versions. In CommitCPG, each node presents a statement with its version information; each edge preserves the semantic level dependency between two statements. To reduce analysis overhead, we perform program slicing [20] over CommitCPGs to remove the nodes/edges irrelevant to the commits. Given a CommitCPG, SCOPY embeds the node statements using CodeBERT [17] and embeds the edge attributes as one-hot vectors to contain the edge versions and the syntax/dependency relationships. Then, a graph convolutional network with multi-head attention is trained over the base and pilot datasets. Finally, we apply SCOPY to identify the security commit candidates from the wild and build the augmented dataset after manual verification.

To enhance the variety of commits beyond the base dataset, we construct the pilot and augmented datasets over popular repositories. We evaluate the efficiency of data collection using the ratio of security commits to the total number of candidates. Compared with random selection, the keyword filtering method and SCOPY can improve the efficiency by 30 and 40 percentage points when constructing the pilot and augmented datasets, respectively. In total, PySecDB contains 1,258 security commits associated with 119 distinctive CWEs across 351 repositories, providing sufficient diversity in vulnerability types and application scenarios. We also find that unique patterns exist in the pilot and augmented datasets, respectively, since these two datasets are built from different perspectives, i.e., commit messages and code changes.

To facilitate software maintenance, we conduct an extensive case study on the security commits in PySecDB and discover four common security fix patterns, i.e., *add or update sanity checks*, *update API usage*, *update regular expressions*, and *restrict security properties*. First, security commits often include sanity checks (i.e., verify if certain conditions are true) to secure critical operations, especially in the authentication and authorization scenarios. Second, since Python provides multiple pre-built packages, security commits can address vulnerabilities by replacing APIs, e.g., APIs related to strings, paths, and commands. Third, security commits can handle secure escapes by updating regular expressions, which protect software from being injected by shell commands, SQL queries, and web scripts. Fourth, security commits can update security properties (e.g., security flags, restriction arguments, and security decorators) to ensure the effectiveness of security mechanisms or policies. These fix patterns can be generalized and formulated into intermediate representations to facilitate secure software development and automated program repair.

```
1  commit dbeb87afefdb63de2f4cff69b6f10c5965d14b54
2  Subject: [PATCH] Fixed code execution bug using
       SafeLoader()
3  diff --git a/pystemon/config.py b/pystemon/config.py
4  @@ -315,7 +315,7 @@ def _load_yamlconfig(self
5  yamlconfig = None
6  ...
7  for includes in yamlconfig.get("includes", []):
8     try:
9         logger.debug("... '{0}'".format(includes))
10 -       yamlconfig.update(yaml.load(open(includes)))
11 +       yamlconfig.update(yaml.safe_load(open(includes)
       ))
12    except Exception as e:
13        raise PystemonConfigException("failed to load
       '{0}': {1}".format(includes, e))
14 return yamlconfig
```

Listing 1: An example of security commit (CVE-2021-27213).

```
1  commit 4cd1067faf3df14dbbe7eb6de2bd7693d5cd829a
2  diff --git a/IPython/lib/security.py b/IPython/lib/
       security.py
3  @@ -109,7 +109,7 @@ def passwd_check(hashed_passphrase
       , passphrase):
4      except ValueError:
5          return False
6  -   if len(pw_digest) == 0 or len(salt) != salt_len:
7  +   if len(pw_digest) == 0:
8          return False
```

Listing 2: An example of non-security commit.

In summary, our paper makes the following contributions:

- We construct the first security commit dataset in Python by screening CVE records and GitHub commits.
- We design a keyword filtering method to identify the potential security commits based on the commit messages.
- Based on code changes, we propose a new commit graph representation CommitCPG and a graph learning model SCOPY to locate the security commit candidates.
- To facilitate software maintenance, we discover four common security fix patterns, which provide insights in vulnerability detection and program repair in Python.

II. Background and Related Work

A. Security and Non-security Commits

On the version control platforms such as GitHub, a commit is mainly composed of two parts: a set of code changes between two versions and a descriptive message including the subject line and body (if any). In Listing 1, Lines 5-14 are the source code changes and Line 2 presents a commit message with only one subject line.

A security commit includes code changes made to a software codebase, addressing a security vulnerability defined by an individual Common Weakness Enumeration (CWE) Specification [21]. Security commits are typically critical updates that need to be applied as soon as possible to prevent attackers from exploiting vulnerabilities. List 1 shows a security commit example fixing the vulnerability CVE-2021-27213 by replacing `yaml.load()` with `yaml.safe_load()` to load the

content in a safer way. Non-security commits are the changes made to the software codebase that do not relate to security issues. These changes include fixing non-security-related bugs, adding new features, improving performance, and updating documentation. Typically, non-security commits are not as urgent as security commits and can be applied later without affecting software security. In List 2, a non-security commit removes the unnecessary check on password salt length.

B. Security Commit Datasets

Security commits provide plentiful information on both the existing vulnerabilities and the corresponding fixes. Thus, researchers construct such datasets for security commit detection and automated program repair [5], [6], [22]–[25]. However, existing datasets only focus on specific projects [6], [15] or contain limited security commits associated with CVEs [22], [24]. Although some researchers also consider both commits indexed by NVD and silent fixes [5], [23], [25], they solely investigate the commits in C/C++ and Java, neglecting the popularity of Python. Besides, the existing works adopt language-dependent security-related features, which cannot be directly migrated for collecting security commits in Python.

C. Security Commit Detection

Numerous commits are submitted to GitHub every day, while 6-10% of them are silent security fixes [7], [26]. To identify security commits automatically and effectively, [27] analyzes the natural language description of commit messages and bug reports. However, this approach only relies on well-maintained documentation, which is impractical for detecting silent security patches. Thus, some researchers detect security commits by extracting code features manually [5], [28]. Wang et al. [9] ensemble two BiLSTM models to learn not only the commit message but also the code changes. Similarly, SPI [15] adopts LSTM to learn the representation of commit message and utilizes CNN to learn the representation of code revision. As the prevalence of applying large language model on code analysis, CodeBERT [17] is fine-tuned to learn the semantics of code changes [12]. Zhou et al. [11] increase the fix data at the function level and then generalize the code change semantic with contrasting learning.

To preserve the inherent structural semantics of code, CLozoya et al. [29] and Wu et al. [14] employ BiLSTM to learn the representation of commits from their AST paths. Wang et al. [13] propose GraphSPD to represent C/C++ commits with code property graphs and learn the representation with GNN.

D. Novelty of Our Study

Dataset. We build the first security commit dataset in Python, which contains 1,258 security commits and 2,791 non-security commits extracted from over 351 popular GitHub projects, covering 119 more CWEs. Different from the SPI [15] based on keyword filtering and PatchDB [5] based on code similarity, we consider both commit message and code changes so that the dataset can cover more diverse security commits, especially for the commits without any clear commit message.

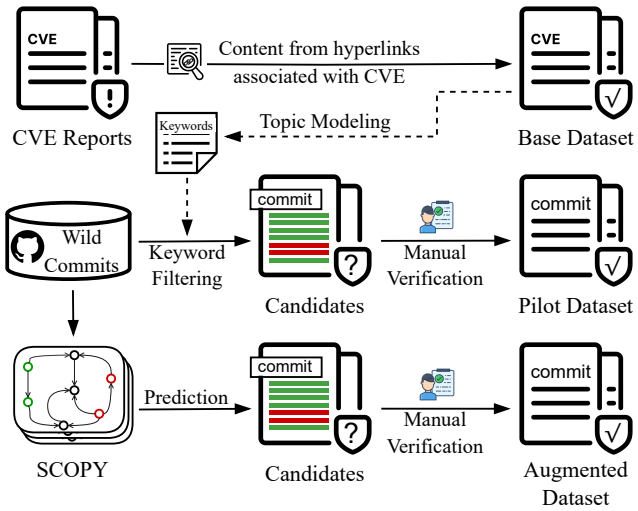

Fig. 1: Overview of collecting three datasets.

Commit Representation. To better represent the commits, we preserve the code structure via the graph representation CommitCPG and also consider the sequential information via CodeBERT [17]. However, the existing works either consider the sequential information [9], [11], [12], [27] or keep the structural semantics [13].

Commit Understanding. We conduct a comprehensive study on understanding how the security commits fix Python vulnerabilities. The commit patterns can be used to generalize vulnerability fix schemes, which may enhance software maintenance and provide insight into automated program repair.

III. Data Collection

Our dataset consists of three sections: *a) base dataset, b) pilot dataset*, and *c) augmented dataset*. Figure 1 illustrates the composition and construction procedure of PySecDB. We first form the base dataset by collecting the commits associated with CVE records indexed by MITRE. Yet, less than 50% of CVE records have published their security commits. To introduce more code semantics, we further build the pilot dataset by filtering the wild GitHub commits that have predefined security keywords in their commit messages. However, not all commits contain well-maintained commit messages that precisely describe the rationales of changed code. Thus, we consider directly mining the most critical part of commits, i.e., the source code changes. To the end, we propose an intermediate commit representation (i.e., CommitCPG) and design a dependency-aggregation graph neural network (i.e., SCOPY) to capture the inherent sequential and structural semantics of code changes. Trained with the base and pilot datasets, SCOPY is able to further build the augmented dataset by pinpointing the silent security commits from the wild.

A. Base Dataset Collection

We build the base dataset according to the CVE records [4]. The first step is to retrieve the vulnerabilities that have already been indexed with CVE IDs. Then, we parse the vulnerability

TABLE I: Security-related keywords for commit filtering.

#Tokens	Keywords
1-gram	attack, bypass, CVE, DoS, exploit, injection, leakage, malicious, overflow, smuggling, spoofing, unauthorized, underflow, vulnerability
2-gram	access control, open redirect, race condition
3-gram	denial of service, out of bound, dot dot slash

reports and crawl the corresponding commits via the provided reference hyperlinks. It comes to our attention that the collected commits may contain some noise, e.g., changelog, test case, refactoring, and renaming. After excluding these unrelated documents, we obtain 729 security commits to form the base dataset; meanwhile, we collect the excluded commits as the non-security subset in the base dataset and expand it by manually identifying the commits that add new features or perform refactoring, linting, and version updates.

B. Pilot Dataset Collection

After examining all the indexed CVE records (as of 01/27/2023), only 46% of them contain the corresponding security fixes. Therefore, the limited samples in the base dataset may not provide adequate syntactic and semantic information. That means, only with the base dataset, we are unable to train a robust model for capturing a wide variety of security commits in the real world. Given the fact that a majority number of security patches are silently committed without reporting to the MITRE [15], [26], we propose to enrich the security commits with the pilot dataset collected from GitHub, i.e., the most common OSS hosting platform.

The pilot dataset is constructed by keyword filtering with humans in the loop. The list of security-related keywords is built automatically by analyzing the CVE descriptions, CWE types (if exist), and commit messages. We determine the final keywords by calculating the word frequency and evaluating the correlation between the keywords and security commits. To obtain the security subset of the pilot dataset, we locate and manually verify the security commit candidates that contain the pre-defined security keywords in the commit messages; the excluded commits are collected as the non-security subset.

Security Keyword Extraction. For each security commit collected from the CVE records, we generate its security impact summary by combining the commit message, the CWE information (if exists), and the CVE report. After generating the summary, we conduct 1-gram, 2-gram, and 3-gram tokenization. Then, we consider the frequency of each token and the correlation of each token with security and non-security commits. We set the frequency threshold and derive the list of security-related keywords, as shown in Table I. Then, we determine the security commit candidates by checking if the wild GitHub commits contain any security keywords in their commit messages. These candidates will be manually verified.

Manual Verification. We hire three security experts to manually verify the security commit candidates. To guarantee the data quality and minimize false positives, the experts are required to follow our two-step labeling procedure strictly. First, each expert tags the commits independently with the labels: security, non-security, or unsure. Then, they gather together to discuss each disagreement and reach a consensus on each uncertain candidate. Only the security commits with 100% agreement will be included in the pilot dataset. In total, it takes 48 man-hours to finish the labeling work.

The proposed keyword filtering mechanism reduces the workload and time for manual verification; meanwhile, the human-in-the-loop ensures the quality of the pilot dataset. More details on labeling efficiency are shown in Section V-A.

C. Augmented Dataset Construction

The pilot dataset overlooks the commits that lack security keywords in the commit messages, while these commits may provide additional variants in syntax and semantics. Therefore, we propose to further augment our dataset. While the pilot dataset is collected based on commit messages, we build the augmented dataset by only analyzing the source code changes. Different from existing works that simply regard source code as sequential data [9], [15], we present a commit graph representation named CommitCPG and a graph learning-based model SCOPY to capture the inherent structural information.

1) CommitCPG: Graph Representation for Commits

To preserve the inherent structure of source code and the modified content between two versions, we propose a graph-based commit representation called CommitCPG, which offers essential syntactic and semantic information for comprehensive commit understanding. Code property graph (CPG) [19] is a program representation that contains abstract syntax trees (AST), control-flow graphs (CFG), and program dependence graphs (PDG), providing a more comprehensive view for code static analysis, compared with traditional sequential structure adopted by NLP-based works [30]. In Figure 2, we first preprocess the raw commits by excluding the irrelevant functions. Inspired by [13], we then merge the CPGs [19] constructed from the previous and current versions by aligning the unchanged statements. Next, we adopt a code slicing method to retain the crucial context-related code snippets, which are not changed directly by commits but can assist us to understand the reason and the effects of code changes.

Commit Preprocessing. To generate the CPG for each code version, we need to retrieve the source code of the previous version and the current version, respectively. To reduce the overhead of CPG generation, we only focus on the modified files instead of the whole project. Then, we extract the functions with code revisions as well as the modified global statements. To achieve this goal, Joern parser [19] is applied to detect all relevant functions and their corresponding scopes. We only retain the functions whose scope overlaps with the modified lines in the commits. For example, in List 1, we will only keep the content of function `_load_yamlconfig()`.

CPG Generation for Previous and Current Versions. With the extracted source code of two versions, we employ Joern [19] to generate the CPGs for both versions.

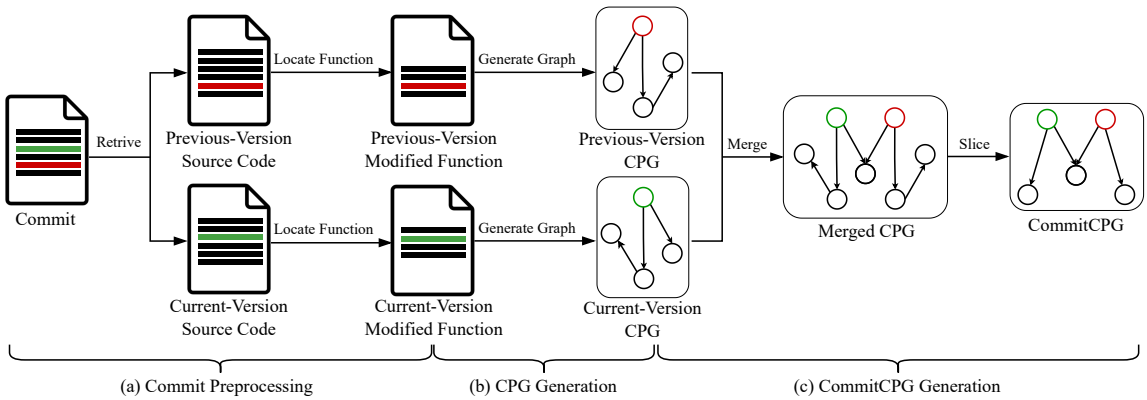

Fig. 2: The generation process of CommitCPGs from commits.

A CPG can be described as (V, E), where V is the node set and E is the edge set. V is comprised of multiple 5-tuples $(id, func\_name, file\_name, version, code)$, which contain the information of each node. The node version, represented by $version \in \{previous, current\}$, reflects if the code line belongs to the previous version or current version. The directed edge set E is made up of 4-tuples $(id_1, id_2, type, version)$, where id_1 and id_2 denote the start and end node IDs, respectively. The edge type, represented by $type \in \{AST, CDG, DDG\}$, specifies if the edge belongs to the AST or control/data dependency graphs. The edge $version$ is consistent with the node's version.

CPG Merging and CommitCPG Slicing. We first generate a unified commit graph by fusing the CPGs of two versions according to each node pair. Then, we update the representation of the merged CPG by two sets: (V', E'). The node set V' is comprised of 5-tuples, which are denoted as $(id, func\_name, file\_name, version, code)$. id is updated so that each node has a unique identifier and the node version is changed to $version \in \{current, previous, unchanged\}$, representing if the code in this node belongs to the current, previous, or both commits. The directed edge set E' is made up of 4-tuples $(id_1, id_2, type, version)$, where id_1, id_2, and $type$ stay the same as E. The edge $version$ will be updated as $unchanged$ if both connected nodes are unchanged nodes.

To reduce the noise introduced by irrelevant nodes and emphasize the semantics of code changes, we generate the CommitCPG by a bi-directional program slicing [20], i.e., forward and backward slicing. Backward slicing is to reason the code changes, while forward slicing is to locate the statements affected by the commit. For example, in List 1, if we set the deleted statement (Line 10) as a backward slicing criterion, the slicing results include Lines 5, 7, and 8; if we set the added statement (Line 11) as a forward slicing criterion, the slicing results contain Line 14. After we conduct code slicing over control/data dependency, we can obtain CommitCPG by only retaining all the nodes of modified and sliced statements (i.e., Line 7, 8, 10, 11, and 14) along with the traced edges.

2) SCOPY: Graph Learning for Commits

Figure 3 illustrates the workflow of SCOPY, our proposed graph-based network to identify security commits in Python. SCOPY contains two steps: (i) node embedding with CodeBERT and edge embedding with dependency-aggregation mechanism, (ii) graph convolution with multi-head attention.

CommitCPG Embedding. To feed the CommitCPG to our SCOPY, we encode the node and edge attributes into numeric vectors. Each node represents a code statement; hence, the node embedding should capture the semantics within the statement. Thus, we first utilize CodeBERT [17] to generate token embeddings and grasp the sequential-based semantics. Then, we obtain the node embedding by aggregating the token embeddings. In addition, each edge presents the dependency between two nodes; thus edge embedding preserves crucial structural and attribute information. We generate edge embeddings with 5-dimensional one-hot vectors. The first two dimensions are used to embed the structural information and present which code version the edge belongs to. The last three dimensions are used to embed the attribute information, which indicates if the edge presents control dependency, data dependency, or syntax relationship.

Graph Convolution with Multi-Head Attention. After embedding the CommitCPG with a sequential model, we adopt a graph convolutional network with multi-head attention to learn the structural representation of commits. To avoid over-smoothing, the number of convolutional layers is limited to 3. We feed the embedded CommitCPG into 3 multi-attributed graph convolutional layers. The node embeddings of CommitCPG are updated with the neighborhood information from different subgraphs. Then, the graph embeddings, i.e., a unified vector representation transformed from all the nodes, edges, and features, can be obtained through graph pooling and vector concatenation. The graph embeddings learned by the SCOPY are finally fed into a multi-layer perceptron to determine the likelihood that a commit fixes a security vulnerability, i.e., whether the given commit is a security commit.

To demonstrate the generalization ability of SCOPY, we utilize the trained SCOPY to discover more security commits in the wild. We directly send the CommitCPG of wild commits into SCOPY. For the commits labeled as security commits, we adopt a similar process (as described in III-B) to manually verify if they are real security commits. The excluded commits composite the non-security subset of the augmented dataset.

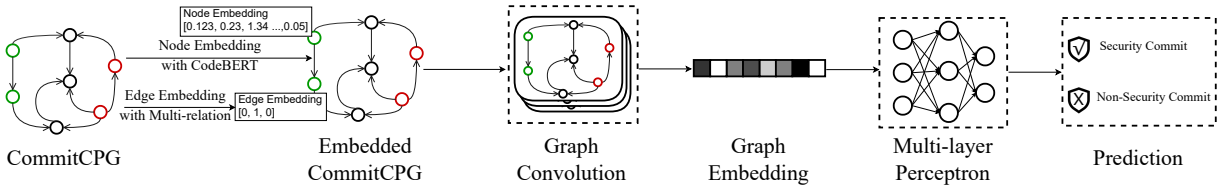

Fig. 3: The workflow of SCOPY that identifies security commits via CommitCPGs.

IV. Implementation

A. Base Dataset Construction

The base dataset consists of the commits linked with CVE records, which have been indexed by MITRE [4]. MITRE provides hyperlinks of vulnerability fixes for 46% of CVE entries. We focus on the hyperlinks from GitHub, where each commit is identified with a unique hash value and the hyperlink is in the form: *https://github.com/{owner}/{repo}/commit/{hash}*. We build the base dataset by downloading the vulnerability fixing commits and removing the commits that are not written in Python or only focus on security-unrelated modifications (e.g., renaming and refactoring).

B. Pilot Dataset Construction

We adopt Latent Dirichlet Allocation (LDA) [18], a topic modeling method, to extract the essential security-related tokens from the commit messages. Then, a keyword filtering algorithm is applied to exclude non-security commits. We download popular open-source repositories in Python and retrieve their commit histories till Jan 27, 2023. For each commit, if it contains at least one proposed keyword, we regard it as a security commit candidate. Later, we manually check these security commit candidates and finalize the dataset.

To facilitate the verification process, we use PyQt [31] to develop a graphical user interface (GUI) that visualizes the code changes of each individual commit and stores the verified security commits according to verification results.

C. Augmented Dataset Construction

To build SCOPY for further dataset augmentation, we first generate the corresponding CommitCPG for each commit in the base and pilot datasets. Specifically, we adopt Joern [19] to generate CPGs for the code versions before and after applying the commit, respectively. Then, we parse the generated graph files and merge the graphs to build CommitCPG. To achieve the program slicing, we develop a Python script to analyze the control/data dependency and AST information and output a sliced CommitCPG ready to be embedded.

To prepare an embedded graph for SCOPY, we embed the nodes and edges respectively. We fine-tune CodeBERT [17] to generate the node embedding dedicated to Python. For the edge embeddings, we apply the one-hot encoding to represent the attributes on each edge. We build the SCOPY on the deep learning library PyTorch 1.6, which is optimized for tensor computing. We develop and optimize our graph model based on the PyTorch-geometric 1.6 library, which supports deep learning on graphs and other structured data. Finally, a multiple-layer perceptron (MLP) is used as a binary predictor, which converts the graph embeddings into predicted labels. We train CommitCPG with the base and pilot datasets. Then, we feed the wild unlabeled commits into the trained SCOPY and apply manual verification to generate our augmented dataset.

V. Analysis Results

After constructing our datasets, we frame our evaluation into four research questions, as outlined below.

- **RQ1:** Can the graph learning-based method help improve the data collection efficiency?
- **RQ2:** How various and representative are the collected security commits?
- **RQ3:** What are the unique patterns of security commits in Python?
- **RQ4:** How do the wild commit samples help improve SCOPY model for downstream security commit detection?

A. Dataset Construction (RQ1)

After keyword filtering and graph-based identification with humans in the loop, we collect 1,258 security commits in total. Specifically, as shown in Table II, there are 729, 400, and 129 security commits in the base, pilot, and augmented datasets, respectively. Also, 2,791 non-security commits are manually labeled during the collection process.

TABLE II: The composition of PySecDB.

Commit \ Dataset	Base	Pilot	Augmented	Total
Security	729	400	129	1258
Non-Security	2134	535	122	2791

Table III lists the augmentation efficiency of random selection, keyword filtering, and SCOPY. Compared with identifying security commits from scratch, the keyword filtering mechanism improves the collecting efficiency by over 30 percentage points and SCOPY improves the efficiency by 40 percentage points.

TABLE III: Efficiency of keyword filtering and SCOPY.

Method	# Candidates	# Verified SC*	Ratio
Random [5]	-	-	6-10%
Keywords	935	400	42.70%
SCOPY	251	129	51.39%

* SC = Security Commits.

TABLE IV: Top 5 repositories by number of security commits.

Repository	#SecurityCommits	Proportion
django	166	13.20%
twisted	87	6.91%
glance	54	4.29%
pillow	41	3.26%
numpy	39	3.10%
Total of Top 5	**387**	**30.76%**

TABLE V: The pattern types of security commits in PySecDB.

Pattern	#Commits	Proportion
1) Add or Update Sanity Checks	416	37.12%
2) Update API Usage	241	19.16%
3) Update Regular Expressions	189	15.02%
4) Restrict Security Properties	183	14.55%
5) Others	178	14.15%
Total	**1258**	**100%**

B. Security Commits Categorization and Distribution (RQ2)

NVD CWE slice [21] associated classification taxonomy serves to identify and describe security vulnerabilities. To understand the purpose of these commits, we investigate the CWE types associated with the CVE reports and plot the distribution of the CWE types that have been explicitly documented. Among the 729 security commits linked to 556 CVEs, due to the limited number of MITRE human analysts, only 312 (56.1%) CVEs have been assigned CWEs. Even so, there are already 119 distinct CWEs associated with our security commits in the base dataset, which means our PySecDB contains at least 119 types of security commits in terms of corresponding vulnerabilities. Figure 4 enumerates the most common CWEs, including frequent security problems such as cross-site scripting (CWE-79), path traversal (CWE-22), etc. Note that we do not directly assign CWE type to security samples in the remaining base, pilot, and augmented dataset since the MITRE CWE team has its own internal process. However, based on our observation and our data collection approaches that are able to introduce wild security commits with more variance (as discussed in V-D), PySecDB can encompass a broad range of security concerns with various kinds of security commits, including but not limited to above 119 CWEs.

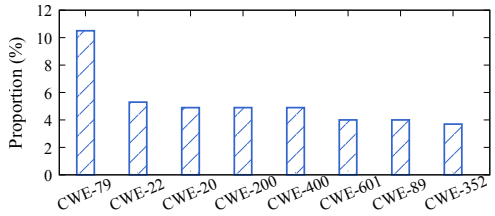

Fig. 4: The top 8 CWE types in PySecDB.

Our collected security commits distribute among 351 popular GitHub repositories unevenly. Among them, 69 repositories provide more than two security commits, bringing a certain amount of variety. In Table IV, the top five repositories that have the most occurrence in our dataset are django [32], twisted [33], glance [34], pillow [35], and numpy [36], implying that the samples in PySecDB align with the popularity trend of security issue in practice.

C. Patch Patterns (RQ3)

We manually go through the whole PySecDB dataset, and discover four common security fix patterns (taking up 85.85% of all security commit samples) that may benefit software maintenance, i.e., adding or updating sanity checks, updating APIs, updating regular expressions, and updating security properties, as listed in Table V.

1) Add or Update Sanity Checks: A sanity check is a basic method to quickly evaluate if a claim or a calculation result can be true, which has been extensively applied to multiple scenarios, e.g., authentication property verification, access control, HTTP request checking [37]. We summarize three representative patterns that fix the vulnerabilities via adding or updating sanity checks, which are presented by 37.12% of security commits in PySecDB.

Authentication. Authentication is the act of proving an assertion, e.g., we need to compare the identity with the system data to verify a system user. The authentication-related vulnerabilities provide attackers the opportunities to masquerade as legitimate users. To defend them, an effective solution is to perform the additional authentication by adding more check requirements or making existing conditions more restrictive. List 3 presents an example of fixing an authentication vulnerability by narrowing down an existing restriction from `True` (i.e., all possible return values except `False`) to `"on"` only.

```
commit 0c0313f375bed7b035c8c0482bbb09599e16bfcf
diff --git a/cps/shelf.py b/cps/shelf.py
@@ -248,7 +248,7 @@ def create_edit_shelf(shelf,
...
          return redirect(url_for('web.index'))
-     is_public = 1 if to_save.get("is_public") else 0
+     is_public = 1 if to_save.get("is_public") == "on"
      else 0
      if config.config_kobo_sync:
...
```

Listing 3: An example of security commit to fix authentication vulnerability (CVE-2022-0273).

Authorization. Authorization refers to the process of granting or denying access to certain data or actions within a system. Authorization comes after authentication and is achieved by an access control list (ACL). The ACL is used to check the user identity with a list of authorized operations and determine which actions a user is allowed to take, e.g., file and data permission. Unrestricted authorization may lead to improper resource consumption since attackers could bypass the system to access high-security level data. List 4 is an example that fixes an authorization bypass exploit by requiring the value of `os.environ.get('GITHUB_ACTIONS')` to be `true`.

HTTP Request. If the interpretation of Content-Length and/or Transfer-Encoding headers between HTTP servers are incon-

```
commit c658b4f3e57258acf5f6207a90c2f2169698ae22
diff --git a/core.py b/core.py
@@ -112,7 +112,7 @@ def actualsys() :
    if attemps == 6:
        ## Brute force protection
        raise Exception("Too many password attempts.")
-   if os.environ.get('GITHUB_ACTIONS') != "":
+   if os.environ.get('GITHUB_ACTIONS') == "true":
        logging.warning("Running on Github Actions")
        actualsys()
    elif uname == cred.name and pwdhash == cred.pass:
```

Listing 4: An example of security commit that fixes an authorization bypass exploit vulnerability (CVE-2022-46179).

sistent, the attackers may take advantage of this issue and send malicious requests to the servers, i.e., HTTP request smuggling. A good solution is to maintain the same interpretation methods in both front-end and back-end servers. In this way, an effective coding practice is to add consistent sanity checks on request interpretation for both servers. List 5 adds such a sanity check on `data` to determine if all characters are digits.

```
commit 8ebfa8f6577431226e109ff98ba48f5152a2c416
diff --git a/src/twisted/web/http.py b/src/twisted/web
    /http.py
@@ -2274,6 +2274,8 @@ def fail():
    if header == b"content-length":
+       if not data.isdigit():
+           return fail()
        try:
            length = int(data)
        except ValueError:
```

Listing 5: An example of security commit that fixes an HTTP request smuggling vulnerability (CVE-2022-24801).

2) Update API Usage: Compared with implementing the fixes from scratch, there are abundant well-formulated packages that can be adopted to realize the intended functionalities and help enforce security restrictions. We notice that a large number (19.16%) of Python security commits fix vulnerabilities by imposing or substituting APIs. We further categorize such security fixes according to their application scenarios.

General Purpose. There is a set of security-related modifications on built-in packages shared by applications for various purposes. For instance, `re.escape` is an API to escape non-alphanumerics that are not part of regular expression syntax, to avoid OS command injection, code injection, and regular expression injection. List 6 is a commit example to fix regular expression injection vulnerability, which demonstrates the application of `re.escape` on `user` and `collection_url`.

```
commit 4bfe7c9f7991d534c8b9fbe153af9d341f925f98
diff --git a/radicale/rights/regex.py b/radicale/
    rights/regex.py
@@ -65,7 +65,10 @@ def _read_from_sections(user,
    collection_url, permission):
...
-       regex = ConfigParser({"login": user, "path":
    collection_url})
+       # Prevent "regex injection"
+       user_escaped = re.escape(user)
+       collection_url_escaped = re.escape(collection_url)
+       regex = ConfigParser({"login": user_escaped, "path
    ": collection_url_escaped})
...
```

Listing 6: An example of security commit that fixes a regular expression injection vulnerability (CVE-2015-8748).

Web Applications. To properly process the inputs of web applications, security commits can adopt APIs in third-party packages for Python (e.g., `parser.quote`, `request.server.escape`, `django.utils.html.escape`, and `html.unescape`) to escape ampersands, brackets, and quotes to the HTML/XML entities or HTTP requests for defeating cross-site scripting (XSS) and HTTP Smuggling. List 7 is an example that fixes an XSS vulnerability by using the API `django.utils.html.escape`.

```
commit f6753a1a1c63fade6ad418fbda827c6750ab0bda
diff --git a/weblate/trans/forms.py b/weblate/trans/
    forms.py
@@ -37,6 +37,7 @@
...
+from django.utils.html import escape
...
-       label = str(unit.translation.language)
+       label = escape(unit.translation.language)
...
```

Listing 7: An example of security commit that fixes an XSS vulnerability (CVE-2022-24710).

Shell Commands. To handle the shell commands securely, security fixes can adopt `shlex.quote` and `subprocess` to load or execute the commands. With the `shlex.quote` API, we can have an escaped version of shell inputs, which can be safely used as tokens in a command line to avoid shell command injection. List 8 is an example that shows the usage of `shlex.quote` to fix a shell injection vulnerability.

```
commit 2817869f98c54975f31e2dd674c1aefa70749cca
diff --git a/canto_curses/guibase.py b/canto_curses/
    guibase.py
@@ -156,6 +156,11 @@ def _fork(self, path, href, text,
     fetch=False):
...
+   href = shlex.quote(href)
...
```

Listing 8: An example of security commit that fixes a shell injection vulnerability (CVE-2013-7416).

Path Name. If a path name is improperly neutralized, attackers may access the files and directories outside of the restricted location. This vulnerability can occur by using absolute file paths or manipulating the path variables where the reference files contain "dot-dot-slash (../)" sequences or variations. To effectively escape such unsafe sequences, Python security commits usually adopt the secure APIs, e.g., `werkzeug.utils.safe_join`, `yaml.safe_load`, and `werkzeug.utils.secure_filename`, to prevent the files or directories from being accessed by malicious users. List 9 is a commit example that fixes a path traversal via using the API `werkzeug.utils.secure_filename`.

3) Update Regular Expressions: Python has become a popular choice for back-end web development, and it is usually combined with some other front-end languages [38]. For this reason, we observe there are 15.02% fixes that modify the regular expressions to avoid XSS, SQL injection, and open redirect vulnerabilities. The regular expression patterns are tailored to match specific strings within the given text, including SQL commands, URLs, and other scripts.

```
commit 1eb1e5428f0926b2829a0bbbb65b0d946e608593
diff --git a/upload/server.py b/upload/server.py
@@ -5,7 +5,7 @@
-
+import werkzeug.utils
@@ -189,7 +189,7 @@ def uploadimage():
    filename = all_files[0][1] + all_files[0][2]
-   remove(filename)
+   remove(werkzeug.utils.secure_filename(filename))
    del all_files[0]
    length = len(all_files)
```

Listing 9: An example of security commit that fixes a path traversal vulnerability (CVE-2022-23609).

SQL Commands. The improper neutralization of SQL commands may lead to SQL injection vulnerabilities, which allow attackers to manipulate the backend database and access the information not intended to be displayed. The corresponding fixes need to escape the unsafe characters. List 10 is a fixed example of SQL injection vulnerability, which substitutes the matched single and double quote characters (i.e., `'` and `"`) in the string `self.queueid`.

```
commit fc2c1ea1b8d795094abb15ac73cab90830534e04
diff --git a/.../model.py b/.../model.py
@@ -772,13 +772,13 @@ def _get_filter(self):
if self.queueid:
-    ... = '%s'" % (self.queueid)
+    ... = '%s'" % (re.sub("[\"']", "", self.queueid))
```

Listing 10: An example of security commit that fixes a SQL injection vulnerability (CVE-2014-125082).

URLs. The improper neutralization of URLs may lead to open redirect vulnerability, which redirects an unsuspecting victim from a legitimate domain to an attacker's phishing site. Effective mitigation is to replace the dangerous special characters with trusted symbols. List 11 is an example of an open redirect vulnerability, which replaces the explicit backslash with an encoded backslash to circumvent the dangerous redirect.

```
commit 08c4c898182edbe97aadef1815cce50448f975cb
diff --git a/auth/login.py b/auth/login.py
@@ -39,6 +39,10 @@ def _redirect_safe(self, url, ...):
+    url = url.replace("\\", "%5C")
     parsed = urlparse(url)
     if parsed.netloc or not (parsed.path + '/').
     startswith(self.base_url):
```

Listing 11: An example of security commit that fixes an open redirect vulnerability (CVE-2019-10255).

Scripts. The improper input validation and encoding during web page generation may lead to XSS, which is able to reveal the cookies, session tokens, or other sensitive information retained by the browser to the attackers. A straightforward solution is to validate the matched characters of a pre-defined pattern. List 12 is an example to fix the XSS vulnerability by re-matching the characters between parentheses instead of the characters between square brackets and validating the matched pattern one by one.

4) Restrict Security Properties: The exploits often result from improper settings of security properties. 14.55% security commits in PySecDB fix improper settings by updating boolean flags from `True` to `False` or vice versa, adding more arguments to methods, or adding security decorators.

```
commit a22eb0673fe0b7784f99c6b5fd343b64a6700f06
diff --git a/helpdesk/models.py b/helpdesk/models.py
@@ -238 +238 @@ def cvesForCPE(cpe,
    if not text:
        return ""
-   pattern = fr'([\[\s\S\]]*?)\(([\s\S]*?):([\[\s\S
    \]]*?)\)'
+   pattern = fr'([\[\s\S\]]*?)\(([\s\S]*?):([\s\S]*?)
    \)'
    # Regex check
    if re.match(pattern, text):
        # get get value of group regex
```

Listing 12: An example of security commit that fixes an XSS vulnerability (CVE-2021-3994).

Update Security Flags. Security flags perform restrictions on the methods that may have access to sensitive objects. Improper restrictions on such flags may expose users to a risky environment and/or lead to sensitive information leakage. List 13 changes the flag from `False` to `True` to fix a vulnerability, where a sensitive cookie does not have a 'HttpOnly' flag.

```
commit 60a3fe559c453bc36b0ec3e5dd39c1303640a59a
diff --git a/src/nsupdate/settings/base.py b/src/
    nsupdate/settings/base.py
@@ -283,7 +283,7 @@
...
-CSRF_COOKIE_HTTPONLY = False
+CSRF_COOKIE_HTTPONLY = True
...
```

Listing 13: An example of security commit that fixes a vulnerability where the sensitive cookie does not have a 'HttpOnly' flag (CVE-2019-25091).

Add Restriction Arguments. Some restriction arguments will be passed to the functions during execution. Improper argument settings may lead to a variety of mishandling. As shown in List 14, the `formaction` is added to restrict the attributes of a variable to avoid XSS vulnerability.

```
commit 10ec1b4e9f93713513a3264ed6158af22492f270
diff --git a/src/lxml/html/defs.py b/src/lxml/html/
    defs.py
@@ -23,6 +23,8 @@
...
+    # HTML5 formaction
+    'formaction'
     ])
...
```

Listing 14: An example of security commit that fixes a cross-site-scripting (XSS) vulnerability (CVE-2021-28957).

Add Security Decorators. A decorator is a function that takes another function and extends the behavior of the function without explicit modification. This mechanism has been widely adopted by security commits to add more detailed security restrictions on existing methods. List 15 shows a security commit that fixes an access control vulnerability by adding decorator `security.private` to function `enumerateRoles`.

D. Unique Patterns Captured from the Wild (RQ4)

Recall that we construct pilot and augmented datasets because the base dataset provides a limited number of security commits samples. Here, we further show the examples captured by our security commit collection approaches that

```
commit 2dad81128250cb2e5d950cddc9d3c0314a80b4bb
diff --git a/src/Products/plugins/ZODBRoleManager.py b
    /src/Products/plugins/ZODBRoleManager.py
@@ -112,6 +112,7 @@ def getRolesForPrincipal(self,
    principal, request=None):
     #   IRoleEnumerationPlugin implementation
+    @security.private
     def enumerateRoles(self, id=None, exact_match=
     False, sort_by=None, max_results=None, **kw):
         """ See IRoleEnumerationPlugin.
```

Listing 15: An example of security commit that fixes an access control vulnerability (CVE-2021-21336).

introduce more variety in syntax and semantics of security-related code changes, enabling wider applications of PySecDB in solving real-world Python-related security issues.

1) Data Variety Introduced by Pilot Dataset: We study the contribution of involving the pilot dataset for SCOPY by comparing the model trained only on the base dataset and the model trained on the combination of the base and pilot datasets. We find that the pilot dataset helps the latter model to be able to identify more wild security commits. For instance, the latter SCOPY can detect more subtle changes. In List 16, the `'%s'` has been changed to `?` in a SQL query, protecting the database from being injected. The capability of detecting such minor changes is enabled by similar samples in the pilot dataset but not existed in the base dataset.

```
commit 9d8adbc07c384ba51c2583ce0819c9abb77dc648
diff --git .../__init__.py .../__init__.py
@@ -71,7 +71,7 @@ def klauen(self,
-    a = u"name == '%s' AND item =='%s'" % (name, item)
+    a = u"name == ? AND item ==?", (name, item)
```

Listing 16: An example of security commit detected by SCOPY trained on the base and pilot datasets.

2) Variance Introduced by Augmented Dataset: We further evaluate to show that our augmented dataset can help train a model that is able to identify more various security commits from the wild. For example, after introducing augmented dataset into the training phase, the model detects a new escape pattern. As shown in List 17, the characters `<`, `>`, and `&` have been escaped by being translated into Unicode, which prevents cross-site-scripting crafted with a partial JSON-serializable object. Compared with the escape expressions in Section V-C that only include ASCII characters, the augmented dataset help SCOPY generalize the escapes to Unicode.

```
commit d3e428a6f7bc4c04d100b06e663c071fdc1717d9
diff --git a/.../djblets_js.py b/.../djblets_js.py
@@ -28,11 +28,18 @@
+_safe_js_escapes = {
+    ord('&'): u'\\u0026',
+    ord('<'): u'\\u003C',
+    ord('>'): u'\\u003E',
+}
```

Listing 17: A security commit example detected by the SCOPY trained on the base, pilot, and augmented datasets.

VI. Discussion

Usability. The SCOPY is versatile and not tied to any specific platform or commit format, making it compatible with a wide range of version control systems like GitHub and GitLab. By integrating the SCOPY as an extension, contributors can easily add vulnerability fix labels to their commits, streamlining the code auditing process and reducing the workload on developers. In addition, downstream developers and users who use third-party libraries can benefit from the SCOPY by being reminded to make necessary security fixes on time. Finally, researchers can obtain labeled commits without requiring extensive manual labor for future data-driven vulnerability and patch-related research.

Ethical Consideration. The SCOPY has the potential to identify undisclosed vulnerability fixes, but this presents a mixed blessing as attackers could exploit this information to target unpatched systems. Our objective in this paper is to prioritize the security of the users' systems; that is why we only share detailed information on the security fixes, rather than the vulnerabilities. By taking this approach, attackers cannot leverage the SCOPY to gain additional details on the vulnerabilities. However, with the SCOPY, open-source software maintainers can quickly reveal vulnerabilities as soon as security fixes become public, improving the overall security of their software systems.

VII. Threats to Validity

Threats to internal validity. Internal bias and errors pose a significant risk in the dataset construction phase. The most essential threat is the exclusion of critical keywords or tokens that are important for identifying security-related commits. To address this issue, we propose an automated approach for learning security-related keywords. However, the current approach of topic models prioritizes the occurrence probabilities of words, which may lead to ignoring infrequent but essential words or tokens. Additionally, commits lacking proper documentation or commit messages are often overlooked, leading to further bias in the dataset.

Threats to external validity. Our experiment and dataset are focused on Python commits, which may limit the generalizability of the SCOPY to other programming languages. Also, since our dataset derives from open-source software, the data may not be applicable for identifying security-related commits in closed-source systems. However, we aim to expand the scope of our research in the future by incorporating more programming languages and diversifying our dataset to enhance the applicability of the SCOPY.

Threats to construct validity. SCOPY is built on top of Joern [39] to construct the code property graphs for the programs of previous and current versions; thus, SCOPY inherits the limitations of Joern. Since Joern disregards the calling relations among multiple functions, SCOPY cannot handle the commits that only change the function calls. Besides, Joern cannot identify the import and use operations of Python packages; thus, SCOPY discards these edges in CommitCPG when the commits only operate packages.

VIII. Conclusion and Future Work

By fully leveraging the commit message and the code change semantics, we construct a large-scale Python security commit dataset named PySecDB that consists of three parts:

base, pilot, and augmented datasets. To enrich the base dataset extracted from CVE reports, the pilot dataset collects the commits that contain the pre-defined security keywords in their commit messages. Given the diversified data samples, we further train SCOPY to learn the security semantics in the code changes to compensate for the poorly documented commits. We conduct a large-scale empirical study of security commits by analyzing PySecDB of 119 CWE categories across 351 repositories. The summarized patterns can assist further software maintenance, e.g., auto program repair. In the future, we will adopt large language models to expand the dataset and apply the summarized patterns to fix the vulnerabilities automatically.

ACKNOWLEDGMENTS

We would like to thank our anonymous reviewers for their valuable comments and suggestions. This work was partially supported by the US Office of Naval Research grants N00014-23-1-2122.

REFERENCES

[1] "TIOBE Index for April 2023." https://www.tiobe.com/tiobe-index/.
[2] J. Ruohonen, K. Hjerppe, and K. Rindell, "A Large-Scale Security-Oriented Static Analysis of Python Packages in PyPI," *2021 18th International Conference on Privacy, Security and Trust (PST)*, p. 1–10, Dec 2021.
[3] PyPI. https://pypi.org/, 2023.
[4] Common Vulnerabilities and Exposures (CVE). https://cve.mitre.org/.
[5] X. Wang, S. Wang, P. Feng, K. Sun, and S. Jajodia, "PatchDB: A large-scale security patch dataset," in *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pp. 149–160, IEEE, 2021.
[6] S. E. Ponta, H. Plate, A. Sabetta, M. Bezzi, and C. Dangremont, "A manually-curated dataset of fixes to vulnerabilities of open-source software," in *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, pp. 383–387, IEEE, 2019.
[7] F. Li and V. Paxson, "A large-scale empirical study of security patches," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pp. 2201–2215, 2017.
[8] S. Kim, S. Woo, H. Lee, and H. Oh, "VUDDY: A scalable approach for vulnerable code clone discovery," in *2017 IEEE Symposium on Security and Privacy (SP)*, pp. 595–614, IEEE, 2017.
[9] X. Wang, S. Wang, P. Feng, K. Sun, S. Jajodia, S. Benchaaboun, and F. Geck, "PatchRNN: A deep learning-based system for security patch identification," in *MILCOM 2021-2021 IEEE Military Communications Conference (MILCOM)*, pp. 595–600, IEEE, 2021.
[10] Z. Li, D. Zou, S. Xu, H. Jin, H. Qi, and J. Hu, "VulPecker: an automated vulnerability detection system based on code similarity analysis," in *Proceedings of the 32nd annual conference on computer security applications*, pp. 201–213, 2016.
[11] J. Zhou, M. Pacheco, J. Chen, X. Hu, X. Xia, D. Lo, and A. E. Hassan, "CoLeFunDa: Explainable Silent Vulnerability Fix Identification,"
[12] J. Zhou, M. Pacheco, Z. Wan, X. Xia, D. Lo, Y. Wang, and A. E. Hassan, "Finding a needle in a haystack: Automated mining of silent vulnerability fixes," in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 705–716, IEEE, 2021.
[13] S. Wang, X. Wang, K. Sun, S. Jajodia, H. Wang, and Q. Li, "GraphSPD: Graph-Based Security Patch Detection with Enriched Code Semantics," in *2023 IEEE Symposium on Security and Privacy (SP)*, pp. 604–621, IEEE Computer Society, 2022.
[14] B. Wu, S. Liu, R. Feng, X. Xie, J. Siow, and S.-W. Lin, "Enhancing security patch identification by capturing structures in commits," *IEEE Transactions on Dependable and Secure Computing*, 2022.
[15] Y. Zhou, J. K. Siow, C. Wang, S. Liu, and Y. Liu, "SPI: Automated identification of security patches via commits," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 31, no. 1, pp. 1–27, 2021.
[16] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.
[17] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, *et al.*, "CodeBERT: A pre-trained model for programming and natural languages," *arXiv preprint arXiv:2002.08155*, 2020.
[18] D. M. Blei, A. Y. Ng, and M. I. Jordan, "Latent dirichlet allocation," *Journal of machine Learning research*, vol. 3, no. Jan, pp. 993–1022, 2003.
[19] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck, "Modeling and Discovering Vulnerabilities with Code Property Graphs," in *2014 IEEE Symposium on Security and Privacy*, pp. 590–604, 2014.
[20] M. Weiser, "Program slicing," *IEEE Transactions on software engineering*, no. 4, pp. 352–357, 1984.
[21] NIST, "NVD CWE Slice." https://nvd.nist.gov/vuln/categories.
[22] J. Fan, Y. Li, S. Wang, and T. N. Nguyen, "AC/C++ code vulnerability dataset with code changes and CVE summaries," in *Proceedings of the 17th International Conference on Mining Software Repositories*, pp. 508–512, 2020.
[23] G. Nikitopoulos, K. Dritsa, P. Louridas, and D. Mitropoulos, "CrossVul: a cross-language vulnerability dataset with commit data," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 1565–1569, 2021.
[24] G. Bhandari, A. Naseer, and L. Moonen, "CVEfixes: 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*, pp. 30–39, 2021.
[25] Y. Chen, Z. Ding, X. Chen, and D. Wagner, "DiverseVul: A New Vulnerable Source Code Dataset for Deep Learning Based Vulnerability Detection," *arXiv preprint arXiv:2304.00409*, 2023.
[26] X. Wang, K. Sun, A. Batcheller, and S. Jajodia, "Detecting" 0-day" vulnerability: An empirical study of secret security patch in OSS," in *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pp. 485–492, IEEE, 2019.
[27] Y. Zhou and A. Sharma, "Automated identification of security issues from commit messages and bug reports," in *Proceedings of the 2017 11th joint meeting on foundations of software engineering*, pp. 914–919, 2017.
[28] A. D. Sawadogo, T. F. Bissyandé, N. Moha, K. Allix, J. Klein, L. Li, and Y. L. Traon, "Learning to catch security patches," *arXiv preprint arXiv:2001.09148*, 2020.
[29] R. Cabrera Lozoya, A. Baumann, A. Sabetta, and M. Bezzi, "Commit2Vec: Learning distributed representations of code changes," *SN Computer Science*, vol. 2, pp. 1–16, 2021.
[30] D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, S. Liu, L. Zhou, N. Duan, A. Svyatkovskiy, S. Fu, *et al.*, "GraphCodeBERT: Pre-training code representations with data flow," *arXiv preprint arXiv:2009.08366*, 2020.
[31] PyQt. https://wiki.python.org/moin/PyQt.
[32] django: The Web framework for perfectionists with deadlines. https://github.com/django/django.
[33] twisted: Event-driven networking engine written in Python. https://github.com/twisted/twisted.
[34] OpenStack Image Management. https://github.com/openstack/glance.
[35] python-pillow/Pillow: Python Imaging Library. https://github.com/python-pillow/Pillow.
[36] numpy: The fundamental package for scientific computing with Python. https://github.com/numpy/numpy.
[37] X. Wang, S. Wang, K. Sun, A. Batcheller, and S. Jajodia, "A machine learning approach to classify security patches into vulnerability types," in *2020 IEEE Conference on Communications and Network Security (CNS)*, pp. 1–9, IEEE, 2020.
[38] Why Use Python for Web Development? https://www.imaginarycloud.com/blog/why-use-python-for-web-development/, Dec 2020.
[39] Joern: The Bug Hunter's Workbench. https://joern.io/.