# *CVEfixes*: Automated Collection of Vulnerabilities and Their Fixes from Open-Source Software

Guru Bhandari
Simula Research Laboratory
Oslo, Norway
guru@simula.no

Amara Naseer
Simula Research Laboratory
Oslo, Norway
amara@simula.no

Leon Moonen
Simula Research Laboratory
Oslo, Norway
leon.moonen@computer.org

## ABSTRACT

Data-driven research on the automated discovery and repair of security vulnerabilities in source code requires comprehensive datasets of real-life vulnerable code and their fixes. To assist in such research, we propose a method to automatically collect and curate a comprehensive vulnerability dataset from Common Vulnerabilities and Exposures (CVE) records in the public National Vulnerability Database (NVD). We implement our approach in a fully automated dataset collection tool and share an initial release of the resulting vulnerability dataset named *CVEfixes*.

The *CVEfixes* collection tool automatically fetches all available CVE records from the NVD, gathers the vulnerable code and corresponding fixes from associated open-source repositories, and organizes the collected information in a relational database. Moreover, the dataset is enriched with meta-data such as programming language, and detailed code and security metrics at five levels of abstraction. The collection can easily be repeated to keep up-to-date with newly discovered or patched vulnerabilities. The initial release of *CVEfixes* spans all published CVEs up to 9 June 2021, covering 5365 CVE records for 1754 open-source projects that were addressed in a total of 5495 vulnerability fixing commits.

*CVEfixes* supports various types of data-driven software security research, such as vulnerability prediction, vulnerability classification, vulnerability severity prediction, analysis of vulnerability-related code changes, and automated vulnerability repair.

## CCS CONCEPTS

• **Security and privacy → Software and application security**; *Vulnerability management*; • **Software and its engineering →** *Software defect analysis*; *Software libraries and repositories.*

## KEYWORDS

Security vulnerabilities, dataset, software repository mining, vulnerability prediction, vulnerability classification, source code repair.

**ACM Reference Format:**
Guru Bhandari, Amara Naseer, and Leon Moonen. 2021. *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 (PROMISE '21), August 19–20, 2021, Athens, Greece.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3475960.3475985

*PROMISE '21, August 19–20, 2021, Athens, Greece*
© 2021 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-8680-7/21/08.
https://doi.org/10.1145/3475960.3475985

## 1 INTRODUCTION

The exploitation of security vulnerabilities is a significant threat to the reliability of software systems and to the protection of the data processed by them, as is frequently evidenced by new data leaks, ransomware attacks, and substantial outages of essential systems. Despite the continued efforts of the software engineering community to improve software quality and security by means of secure coding guidelines, software testing, and various forms of code review, the publicly available Common Vulnerabilities and Exposures (CVE) records reveal an increasing trend in the number of vulnerabilities that are discovered each year [1].

The overall security of software is highly dependent on the effective and timely identification and mitigation of software vulnerabilities. However, this is not an easy task and requires experience and specialized skills that go beyond the expertise of the typical developer, resulting in many vulnerabilities that go unnoticed for a long time. Consequently, there has been considerable attention in academia and industry to the development of techniques and tools that can help developers identify, and possibly repair, security vulnerabilities in source code already in the development phase.

*Data-driven* vulnerability research depends on the availability of datasets with samples of *real-life* vulnerable code and their fixes [2]. Moreover, such datasets should encompass multiple levels of *granularity* [3, 4], such as files, classes, functions, etc., and cover widely-used programming languages [5, 6]. Finally, for reliable training and evaluation of machine learning (ML) approaches, we need *comprehensive* datasets that contain large numbers of diverse and *labeled* samples of both vulnerable and non-vulnerable code [7–9]. As we will see in the discussion of related work (Section 2), the currently available vulnerability datasets do not fulfill these requirements.

To address their shortcomings and assist in data-driven vulnerability research, we propose to automatically collect and curate a comprehensive vulnerability dataset from CVE records in the National Vulnerability Database (NVD). In particular, we propose an approach that *mines* CVE records for open-source software (OSS) projects hosted on GitHub, GitLab, and Bitbucket, to collect real-world samples of vulnerable and corresponding patched code. We implement the proposed approach in a dataset collection tool and share an initial release of the resulting vulnerability dataset named *CVEfixes*. The dataset supports various types of data-driven software security research, such as ML-based vulnerability identification, automated classification of identified vulnerabilities in Common Weakness Enumeration (CWE) types, vulnerability severity prediction, and automated repair of security vulnerabilities.

**Contributions:** This paper makes the following contributions:

(a) We survey existing security vulnerability-related datasets and discuss their strengths and weaknesses.

(b) We propose a method to automatically collect and curate a comprehensive vulnerability dataset from CVE records in the NVD, and obtain real-world samples of vulnerable and corresponding patched code, organized at multiple levels of granularity. In addition, we discuss how to enrich this data with meta-data such as programming language used, and detailed code-related metrics at five levels of abstraction, such as the commit-, file-, and method levels, as well as the repository- and CVE levels.
(c) We implement the proposed approach in a dataset collection tool that is made publicly available.[1]
(d) We publicly share a version of the *CVEfixes* vulnerability dataset for use by other researchers.[2] The initial release spans all published CVE records up to 9 June 2021, covering 5365 CVE records for 1754 OSS projects. A total of 5495 vulnerability fixing commits are obtained from the projects' version control systems and linked to information from the corresponding CVE records, such as CVE-IDs, reference links, severity scores, vulnerability type/CWE type, and other descriptive information.

The remainder of the paper is organized as follows: Section 2 discusses the currently available vulnerability datasets and how our work differs from them. Section 3 presents the process for constructing the *CVEfixes* dataset, followed by an exploration of the initial release in Section 4. Section 5 discusses applications of the dataset, together with current limitations and future extensions. Finally, we conclude in Section 6.

## 2 RELATED WORK

Several vulnerability-related test suites and datasets were developed in the last decade, often with particular goals in mind. These sets were consequently picked up by other security researchers looking for relevant data to train or evaluate their techniques on. This section surveys frequently used datasets and discusses some of the challenges for their reuse in other contexts. We also look at more general software repositories and repository mining frameworks. Finally, we discuss how the *CVEfixes* dataset is different from the existing work in this area.

**Vulnerability Datasets** The Software Assurance Metrics And Tool Evaluation (SAMATE) project created the Juliet Test Suite as a benchmark for static analysis tools that aim to identify vulnerabilities in source code [10, 11]. Programs from the Juliet Test Suite have been used in several studies on vulnerability and weakness prediction [12–16]. However, the benchmark was never created for this purpose, and concerns have been raised that the vulnerabilities in Juliet are not very diverse, and many of them are of a synthetic nature that does not occur in real-world software projects [2, 17].

Mitropoulus et al. [18] constructed a vulnerability dataset by analyzing the Maven repository using *FindBugs* [19], a tool conducting static code analysis on Java byte-code, identifying the vulnerabilities in Maven and categorizing them into nine different types. Similarly, Draper's VDISC dataset [7] consists of C/C++ source code of 1.27 million functions mined from open-source projects that were labeled by three static code analysis tools (Clang, Cppcheck, and Flawfinder) as vulnerable or not, classified in five groups of vulnerabilities, namely CWE-120, 119, 469, 476, and other. However, there are some challenges with the dataset, such as the fact that it is highly imbalanced with only 6.8% functions labeled as vulnerable. Moreover, the extracted functions are incomplete, missing the function's return type which excludes certain (signature-based) analyses. *SVCP4C* (*SonarCloud* Vulnerable Code Prospector for C) is an online tool for collecting vulnerable source code from open-source repositories linked to SonarCloud. The tool performs static analysis and labels the potentially vulnerable source code at the file level [20]. The Devign [21] dataset includes four real-world open-source C/C++ projects, Linux, FFmpeg, Qemu and Wireshark, where the labeling is performed using security-related keyword filtering. The drawback of this dataset is also in the labeling: if a commit is believed to be vulnerable, then all functions changed by the commit are labeled as vulnerable, which is always not true. Moreover, the dataset does not classify the types of vulnerabilities encountered. A threat recognized by the authors of these datasets/-tools is that the labeling is based on static analysis, which is known for its false positives, that may lead to incorrect labeling. Our approach, as well as others discussed below, aims to mitigate this threat by building on actual security patches. The Code Gadget Database (CGD) [22] collects two types of vulnerabilities in C/C++ programs: buffer error vulnerability (CWE-119) and resource management error vulnerability (CWE-399). The dataset covers 61638 code gadgets including 17725 vulnerable and 43913 non-vulnerable code gadgets. It aims to improve the labeling quality by checking if the code slice of confirmed bugs overlaps with a bug fix before labeling code as 'buggy'. Recently, Zheng et al. [17] published *D2A*, which uses an approach based on differential analysis to label issues in the source code functions or snippets reported by static analysis tools. The D2A approach analyzes the commit messages to identify the likeliness of vulnerability fixes from before-commit and after-commit versions. The dataset considers code from six open-source C/C++ projects: OpenSSL, FFmpeg, httpd, NGINX, libtiff, and libav.

Vieira et al. [23] introduced a dataset of bug-fixing activities from 70000 bug-fixing reports from 10 years of 55 open-source projects of *Apache* mined from *Jira*,[3] a popular issue tracking system that captures information about software development, bugs, security vulnerabilities, new functionalities, etc. The dataset's emphasis is on process-related information regarding issue management, such as change metrics, fix effort, status, version, and assignee, but it does not go into code-level detail. Alves et al. [24] analyze the bug-tracking systems of five open-source projects (Mozilla, Linux Kernel, Xen Hypervisor, Httpd, and Glibc), looking for occurrences of CVE identifiers. They found 2875 security patches and used the associated code to build a dataset that labels the code before patching as vulnerable, and after patching as not vulnerable, as well as adding labels for vulnerability types and severity. Moreover, they compute software metrics at the file, class, and function level for the code before and after patching to enable metrics-based vulnerability prediction research. Similarly, Gkortzis et al. [25] present a vulnerability dataset correlating source code and software metrics of 8694 versions of open-source projects.

[1] https://github.com/secureIT-project/CVEfixes, DOI:10.5281/zenodo.5111494.
[2] https://zenodo.org/record/4476563, DOI:10.5281/zenodo.4476563.
[3] https://atlassian.com/software/jira

Jimenez et al. [26] present *VulData7*, a framework to collect a security vulnerability dataset covering all reported NVD vulnerabilities of four security-critical open-source systems, *Linux Kernel, WireShark, OpenSSL, and SystemD.* Although focusing on a limited set of systems all written in the C programming language, their approach shares several characteristics with ours, but one main practical challenge for reusing their framework is that it is based on the analysis of XML files, a format that is no longer provided by the NVD. Similarly, Ponta et al. [27] curated a dataset of 1282 commit fixes to vulnerabilities from 205 open-source Java projects obtained from NVD and project-specific web resources, and Fan and Nguyen [28] curated a C/C++ code vulnerability dataset named *Big-Vul* that corresponds with code changes and CVE database entries from 2002 to 2019.

Lin et al. [9] emphasize the need for a benchmark for evaluating the effectiveness of ML approaches aimed at fixing security vulnerability in source code. The authors propose a benchmark dataset that offers labels at two levels of granularity, i.e., the file- and method level, collected from nine software projects written in the C programming language [29]. The challenge with using this dataset for training purposes is that it contains only 1471 vulnerable functions and 1320 vulnerable files, which is rather small for training a deep learning model.

**Software Repositories and Repository Mining Frameworks** Ma et al. [30] present World of Code (WoC), a large and frequently updated collection of git-based version control data. The data is indexed three storage abstractions and arranged in four layers. New and updated repositories are periodically analyzed, and the database is updated once a month. The dataset stores four types of raw git objects: commits, trees, blobs and tags. The bulk extraction of these raw git objects was done in parallel using *libgit2*, a portable tool implemented in C. Querying this database can help answer different research questions, such as trends in programming languages used, ecosystem comparisons, bug prediction, developer migration studies, bot detection, understanding developer trajectories, etc. In the same way, the dataset can be inspected for different security-related research areas, vulnerable commits classification, developers' tendency towards vulnerability, etc.

*Perceval* [31] is an automatic and incremental data gathering tool to mine software development data from various sources such as versioning systems, bug tracking systems and mailing lists. The JSON output of the tool stores commit-level information of the repositories that can be used to analyze the activities of the repositories and the developers. However, the tool does not expose access to the finer granularity levels of open-source projects, such as the code-level vulnerability information that we are concerned with.

*PyDriller*[4] is a Python package to mine code-level information from different version control systems [32]. It allows extracting commit-, file-, method levels information from git repositories, and it collects all relevant meta-information, including the computation of a number of code-level metrics. Because of these features, we use *PyDriller* during the construction of *CVEfixes*, before mapping the collected data to a relational database.

**Differences** The *CVEfixes* dataset presented in this paper differs from the existing datasets in the following ways: (1) In general, it covers more information about the vulnerability at different levels of granularity, up to the actual source code before and after fixing. (2) It covers a longer timeframe, being based on all CVE records available at the collection date of 9 June 2021. Moreover, it's not restricted to this timeframe: at any point in the future, our publicly available tool can automatically collect and integrate the data up to the last published CVE at that point in time. (3) Unlike many other datasets that focus on specific programming languages like Java or C/C++, our dataset covers multiple programming languages and allows the language as a query attribute. (4) Some of the existing vulnerability datasets only classify the source code as vulnerable or not. In addition to this binary classification, *CVEfixes* classifies the vulnerabilities into categories as defined by Common Weakness Enumeration (CWE) types, as well as CVSS severity scores, enabling research into multi-class vulnerability prediction.

## 3 DATASET CONSTRUCTION

### 3.1 Overall Workflow

The National Vulnerability Database (NVD) [33] is a repository of vulnerability management data maintained by the National Institute of Standards and Technology (NIST). NIST publishes the entire NVD database for public use employing various data feeds, including JSON vulnerability feeds, security checklist references, security-related software flaws, misconfigurations, product names, and impact metrics. The JSON vulnerability feed comprises a daily updated feed of vulnerability records (CVEs), organized by year of origin (updates may concern CVEs published in earlier years).

Each vulnerability in the file has its CVE-ID, publish date, description, associate reference links, vulnerable product configuration, CWE weakness categorization [34], and other metrics. Moreover, each vulnerability's severity is ranked using the Common Vulnerability Scoring System (CVSS) [35]. The CVSS is composed of a set of metrics in three groups, i.e., base, temporal, and environmental, to specify the characteristics and contextual information of a vulnerability. There are two versions of the scoring system: $\text{CVSS}_{\text{v2}}$ and $\text{CVSS}_{\text{v3}}$, that are both in active use in the NVD.[5] $\text{CVSS}_{\text{v3}}$ introduces a number of changes over $\text{CVSS}_{\text{v2}}$ to score vulnerabilities more accurately and provide more information to distinguish between different types of vulnerabilities. Finally, when a vulnerability in an open-source project is fixed, the CVE record will be updated with one or more pointers to the relevant source code repositories, as well as commit hashes of the fixes. These comprehensive, scrutinized, and frequently updated JSON vulnerability feeds are the primary basis for generating the vulnerability data in *CVEfixes.*

Figure 1 presents the overall workflow of the dataset collection process. It starts by collecting CVE records via JSON vulnerability feeds. Since these CVE vulnerabilities are categorized according to CWE weakness types [34], we also collect the details of these CWE types and cross-reference them with the appropriate CVE records.

Moreover, for CVEs that have fixes associated with them, the corresponding open-source project repositories are (temporarily) locally cloned, and source-level information about the vulnerable code and the corresponding fixes is gathered based on the commit hashes reported in the CVE record. After extracting the vulnerable

[4] https://github.com/ishepard/pydriller

[5] https://nvd.nist.gov/vuln-metrics/cvss

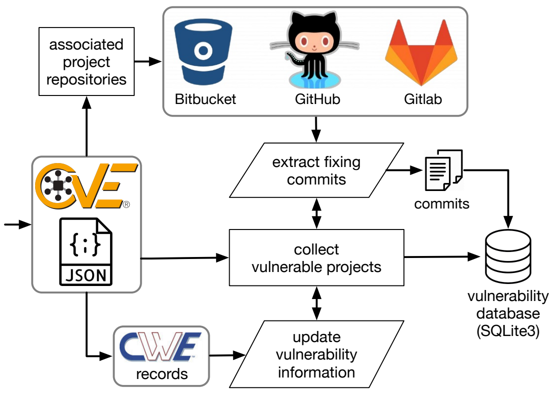
associated project repositories
Bitbucket
GitHub
Gitlab
extract fixing commits
commits
collect vulnerable projects
vulnerability database (SQLite3)
JSON
records
update vulnerability information

**Figure 1: Dataset construction workflow**

and fixed code, a number of code-level metrics are computed, and additional meta-data such as the programming language is derived. Finally, the collected information is stored in a relational database.

The overall structure of the database is shown in Figure 2. The main tables capture information about CVE records, CWE records, OSS repositories, commits, changed files, and changed methods. Two additional tables respectively associate fixes with CVEs, and CWE classifications with CVEs.

## 3.2 Details of the Automated Collection Tool

The proposed process has been implemented in a fully automated *CVEfixes* collection tool, and the remainder of this section discusses the major phases implemented in the tool. The collection can easily be repeated to keep up-to-date with newly discovered or patched vulnerabilities. Since updates can affect any of the published CVEs, the collection tool currently revisits all published CVEs to check if relevant fixes were added. One direction for future work is enabling a more incremental process.

**Scanning of CVE Records:** By default, the collection tool retrieves all published JSON vulnerability feeds from the NVD server, covering the first published CVE in 2002 up to the last published one on the date of collection (the feeds are updated daily). The JSON files are then aggregated, flattened, and processed to filter out redundant information from the CVE records (in the original feeds, some fields are repeated at deeper nesting levels). All CVE records that do not have fixes associated with them in the *reference_json* field are ignored, because gathering corresponding vulnerable or fixed code is not possible for these cases. After processing and filtering the records, various details of the vulnerability such as CVE-ID, published date, last modified date, reference data, CVSS severity scores, vulnerability impact and scope, exploitability score, are recorded in the *cve* table.

**Classification of Vulnerabilities:** As discussed before, CVE records in the NVD are annotated with CWE vulnerability types that indicate at a more abstract level what kind of vulnerability this CVE concerns. The *problemtype_json* field of the NVD JSON vulnerability feeds may refer to one or more of these CWE types. The NVD only uses a subset[6] of the full CWE list maintained by MITRE,[7] and does not distinguish between individual CWEs and CWE categories, referring to either as "CWE-<*num*>". Moreover, two additional classifications, "*NVD-CWE-noinfo*" and "*NVD-CWE-other*", are used to refer to cases where, respectively, there is not enough info for classification, or where the NVD deviates from the full CWE. Note that we also found around 250 cases where the NVD feeds contain the CWE classification "*unknown*". For consistency, we map these "*unknowns*" to "*NVD-CWE-noinfo*" in our dataset.

**Meta-Information for Repositories:** At the time of writing, the majority (>80%) of fixes in the NVD refer to code on one of the three major OSS forges, GitHub, GitLab, and Bitbucket, with 98% of those being on GitHub. The remaining 20% of fixes point to other forges (e.g., sourceforge or the defunct gitorious), or to project-specific servers hosting other versioning systems (e.g., mercurial, subversion, or CVS). As a result, we focus on gathering code from those three OSS forges, and we have modeled the *repository* table to maximally cover GitHub's repository meta-information, such as repository name, description, date of creation, last date of push, homepage, programming language, number of forks and number of stars. This information can be used to, for example, focus on certain programming languages or filter for popular repositories by setting a minimum threshold on the number of stars. The latter is also a common and effective approach to focus on information related to *mature* projects. Note that GitLab and Bitbucket do not offer all of these attributes, so clearly distinguishable values are used to represent missing information, such as the value "-1" for the *stars_count* of Bitbucket repositories (which does not offer stars).

**Extraction of Commits:** Whenever an OSS vulnerability is fixed, the *reference_json* in the CVE record contains the repository URL, as well as a pointer to the exact commit that introduced the fix, by means of a git hash. We use this information to locally (and temporarily) clone the repository, and use it to extract versions of the code before and after submitting the fix. We extract this information at the commit-, file- and method level, where each entry in the *commits* table is associated to one or more *cve*s via the *fixes* table. In addition, the *commits* table stores meta-information about the commit, such as the author, time and date, commit message, if it is a merge commit (often to include pull requests in projects using GitHub Flow), the number of lines added or deleted, and Delta Maintainability Model metrics related to the change [36].

**Extracting the Modified Files:** Every entry in the *commits* table is linked via the commit hash to one or more *file_change*s that were included to fix the vulnerability (or vulnerabilities) associated with the commit. For each file change, the dataset contains the contents of the file before and after making the change, in *code_before* and *code_after* respectively, as well as the *diff* of the change in the format delivered by Git, and a parsed version of this information in *diff_parsed*, containing a dictionary of added and deleted lines. In addition, several meta-data are collected, such as filename, old and new path, type of the modification (i.e., added, deleted, modified, or renamed), number of lines added or removed in that file, lines of code (*nloc*) after the change, and cyclomatic complexity after the change. Finally, the *Guesslang* tool[8] is used to detect the actual *programming_language* that is used in a given file. *Guesslang* is an

[6] https://nvd.nist.gov/vuln/categories

[7] https://cwe.mitre.org/data/index.html

[8] https://guesslang.readthedocs.io/en/latest/

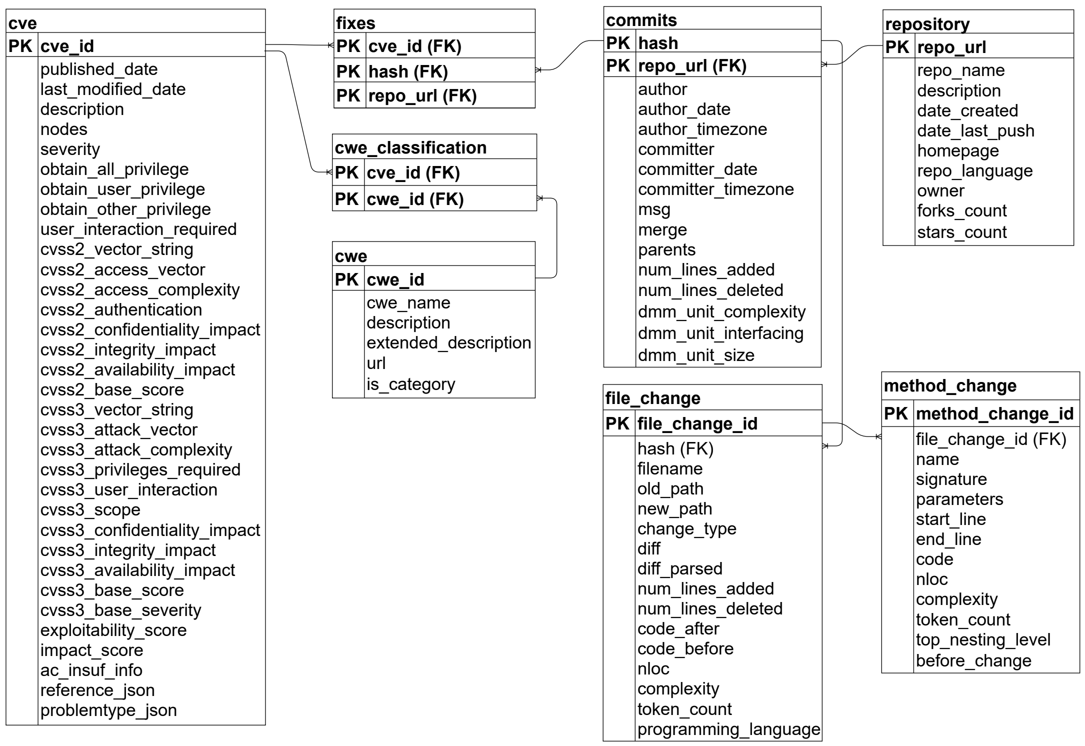

**Figure 2: Entity-Relationship Diagram showing the various levels of abstraction in *CVEfixes* and their interconnections**

open-source deep-learning based classifier that was trained with over a million source code files, and recognizes over 30 programming languages. Note that these detected languages may differ from, and are more precise than the repository language that is, for example, reported by GitHub.

**Extracting the Modified Methods:** Similar to the changes at the file-level, we keep track of changes at the method-level in the *method_change* table. In addition to the actual code, each method change stores meta-information such as the method name, its signature, parameters, start and end line of the method, lines of code, cyclomatic complexity, and a boolean indicating whether this concerns the code from before committing the fix or not.

## 3.3 Practical Guidance

The automated *CVEfixes* dataset collection tool is distributed via GitHub.[9] The distribution comes with detailed requirements and installation instructions. It requires Python3.8 or later to run, and SQLite3 to construct and store the data into a relational database. The main Python packages that it depends on are pandas, numpy, requests, PyDriller, PyGithub and guesslang. We provide *requirements.txt* and *environment.yml* files for dependency management using pip/venv or anaconda/miniconda.

The tool is configured by means of a number of settings in a *.CVEfixes.ini* file (an example is provided in the distribution):

- *database_path*: location of the *CVEfixes* database file (also used to hold some temporary files during extraction).
- *sample_limit*: an optional limit on the number of commits to be gathered (mainly for testing and demonstration purposes, sample_limit = 0 means unlimited collection).

Note that the GitHub API is severely rate-limited when unauthenticated access is used. These limits can be raised significantly to up to 5000 requests per hour by authenticatinsg with a username and personal access token.[10] A *github_username* and *github_token* can be configured in *.CVEfixes.ini* for this purpose. With a sample limit of 25, no token is needed and rate-limiting will not be triggered.

We provide a compressed SQL dump for the initial release of the *CVEfixes* vulnerability dataset that covers all published CVEs in the NVD up to 9 June 2021 at Zenodo.[11] The distribution contains a simple shell script to convert the compressed SQL dump into an SQLite3 database:

```
$ sh Code/create_CVEfixes_from_dump.sh
```

Alternatively, the *CVEfixes* dataset can be gathered from scratch using the following shell script:

```
sh Code/create_CVEfixes_from_scratch.sh
```

Note that this process will overwrite an existing database. Moreover, at the time of writing, the full collection process can take up to 15 hours, depending on the available internet connection. The advantage of taking this route is that the database will contain all CVEs published up to the date of initiating the collection process.

[9] https://github.com/secureIT-project/CVEfixes, DOI:10.5281/zenodo.5111494.

[10] https://docs.github.com/en/github/authenticating-to-github/creating-a-personal-access-token

[11] https://zenodo.org/record/4476563, DOI:10.5281/zenodo.4476563.

The distribution contains an example Jupyter Notebook *Examples/query_CVEfixes.ipynb* that shows how to explore the *CVEfixes* dataset and contains the code to generate the various tables and graphs in this paper.

Recall the overall structure of the database shown in Figure 2. Then the following query extracts all methods involved in fixes related to C programs before the changes were made:

```
SELECT m.name, m.signature, m.nloc, m.
    parameters, m.token_count, m.code
FROM method_change m, file_change f
WHERE f.file_change_id = m.file_change_id
AND f.programming_language = 'C'
AND m.before_change = True
```

Another example, inspired by the ManySStuBs4J dataset [37], to find all files that only add or remove a single line as fix:

```
SELECT cv.cve_id, f.filename, f.
    num_lines_added, f.num_lines_deleted, f.
    code_before, f.code_after, cc.cwe_id
FROM file_change f, commits c, fixes fx, cve
    cv, cwe_classification cc
WHERE f.hash = c.hash AND c.hash = fx.hash
AND fx.cve_id=cv.cve_id
AND cv.cve_id=cc.cve_id
AND f.num_lines_added<=1
AND f.num_lines_deleted<=1;
```

## 4 DATASET EXPLORATION

Table 1 presents the summary statistics of the initial release of the *CVEfixes* dataset. This initial release covers 5365 unique CVEs in 1754 OSS projects, with 5495 unique vulnerability fixing commits. The CVEs are classified into 180 different CWE vulnerability types. Note that some of the vulnerability fixing commits cover multiple CVEs, as is indicated by the number of commits being lower than the number of CVEs.

To further characterize the data, Table 2 presents the top ten projects in *CVEfixes* with respect to the number of CVEs, number of vulnerability fixing commits, and number of files and methods involved in fixes. In particular, Table 2a corresponds with the project-wise number of CVEs, the average $\text{CVSS}_{\text{v2}}$ and $\text{CVSS}_{\text{v3}}$ base scores of these vulnerabilities, as well as their average exploitability, and impact scores. Not surprisingly, the long-running and well-scrutinized *linux* project has the most vulnerabilities and vulnerability fixing commits (shown in resp. Table 2a and 2b), i.e., 1029 commits for fixing 973 vulnerabilities. Note that *linux* has roughly six times as much vulnerabilities as *ImageMagick*, the project with second most vulnerabilities (157), yet the security impact of those discovered in *ImageMagick* is considerably larger, as shown by the various security metrics.

**Table 1: Summary statistics of the *CVEfixes* dataset**

| CVEs | CWEs | projects | commits | files | methods |
|---|---|---|---|---|---|
| 5,365 | 180 | 1,754 | 5,495 | 18,249 | 50,322 |

**Table 2: Top 10 projects in *CVEfixes* with respect to (a) the number of fixed CVEs, shown with the project's security metrics, (b) number of vulnerability fixing commits, and the number of (c) files and (d) methods involved in these fixes.**

| project | # CVEs | average $\text{CVSS}_{\text{v2}}$ | average $\text{CVSS}_{\text{v3}}$ | average exploitability | average impact |
|---|---|---|---|---|---|
| linux | 973 | 5.30 | 3.99 | 1.16 | 2.78 |
| ImageMagick | 157 | 5.45 | 7.25 | 2.74 | 4.43 |
| tensorflow | 143 | 3.92 | 6.67 | 2.05 | 4.50 |
| tcpdump | 89 | 7.33 | 9.57 | 3.86 | 5.70 |
| FFmpeg | 83 | 6.35 | 5.86 | 2.23 | 3.57 |
| phpmyadmin | 67 | 4.46 | 3.55 | 1.85 | 1.53 |
| php-src | 60 | 6.59 | 7.46 | 3.22 | 4.21 |
| MISP | 50 | 5.03 | 6.70 | 2.81 | 3.61 |
| WordPress | 46 | 5.27 | 7.01 | 3.06 | 3.66 |
| hhvm | 40 | 6.48 | 7.54 | 3.22 | 4.31 |

(a)

| project | # commits |
|---|---|
| linux | 1,029 |
| ImageMagick | 171 |
| tensorflow | 156 |
| phpmyadmin | 126 |
| tcpdump | 101 |
| FFmpeg | 88 |
| php-src | 61 |
| MISP | 54 |
| WordPress | 47 |
| radare2 | 43 |

(b)

| project | # files |
|---|---|
| linux | 2,007 |
| GeniXCMS | 1,930 |
| kanboard | 698 |
| CycloneTCP | 611 |
| X2CRM | 594 |
| exponent-cms | 445 |
| Ushahidi-Web | 402 |
| wityCMS | 372 |
| tcpdump | 344 |
| tensorflow | 340 |

(c)

| project | # methods |
|---|---|
| GeniXCMS | 14,235 |
| linux | 4,303 |
| exponent-cms | 3,048 |
| Ilch-2.0 | 1,427 |
| arangodb | 1,224 |
| kanboard | 1,169 |
| tensorflow | 1,045 |
| moped | 950 |
| WordPress | 915 |
| X2CRM | 790 |

(d)

Looking at the projects in the four top ten lists of Table 2a-2d, it is not surprising to see some overlap between the lists. On the other hand, it *is* surprising to see that some projects, such as *GeniXCMS*, *kanboard*, and *exponent-cms*, which do not occur at all in the top ten CVEs and #commits, come in very highly ranked on the amounts of files and methods that need to be changed to fix vulnerabilities in these projects. This sudden rise in the ranks suggests it can be of interest to investigate how the modularization decisions in these projects impacted the security fixes that were made, and may indicate an application-level code smell such as shotgun surgery.

This finding also shows that there is *no* direct correlation between the number of files or methods and the number of CVEs (or commits to fix them), because that would mean the respective top tens would have been similar. We *do* see the expected relation between CVEs and vulnerability fixing commits (with minor changes due to some commits covering multiple CVEs, as noted earlier).

The 10 projects in Table 2a cover 31.84% of the total number of CVEs, those in Table 2b cover 31.61% of the total number of vulnerability fixing commits, the ones in Table 2c account for 42.43% of the total number of files involved in fixes, and finally, the ten projects in Table 2d make up 57.84% of the total number of methods involved in fixes.

Table 3 presents the ten most occurring CWE types based on CVE count. We see that CWE-79 (Improper Neutralization of Input During Web Page Generation, is the most commonly identified

**Table 3: Distribution of vulnerabilities over CWE types**

| CWE | description | CVEs | cmts | files |
|---|---|---|---|---|
| CWE-79 | Improper Neutralization of Input During Web Page Generation (Cross-site Scripting) | 635 | 670 | 3226 |
| CWE-119 | Improper Restriction of Operations within the Bounds of a Memory Buffer | 408 | 403 | 716 |
| CWE-20 | Improper Input Validation | 382 | 397 | 1965 |
| CWE-125 | Out-of-bounds Read | 380 | 404 | 1061 |
| CWE-200 | Exposure of Sensitive Information to an Unauthorized Actor | 276 | 314 | 818 |
| CWE-787 | Out-of-bounds Write | 205 | 211 | 537 |
| CWE-476 | NULL Pointer Dereference | 195 | 198 | 334 |
| noinfo* | Insufficient Information | 193 | 219 | 664 |
| other* | Other | 165 | 170 | 395 |
| CWE-264 | Permissions Privileges and Access Controls | 143 | 148 | 457 |

* abbreviated from NVD-CVE-noinfo and NVD-CWE-other for space.

vulnerability type. This type is commonly known as Cross-site Scripting or XSS. It has been assigned to 635 CVEs and was fixed in 670 commits that changed 3226 files. Observe that the number of vulnerability fixing commits is larger than the number of CVEs, which is caused by the fact that some of the CVEs are associated with multiple fixes. This occurs, for example, when a CVE is fixed in several related projects, or when the fix is spread over multiple commits. On second place comes CWE-119 (Improper Restriction of Operations within the Bounds of a Memory Buffer), reported in 408 CVEs and fixed in 403 commits that changed 716 files.

Next, we find CWE-20 (Improper Input Validation) assigned to 382 CVEs. which is closely followed by CWE-125 (Out-of-bounds Read), which is actually a "child" of the more general CWE-119 category, and indicates that an index pointer is used beyond the bounds of the memory buffer. It is reported in 380 CVEs and fixed in 404 commits that modified 1159 files. After this top four, we see a considerable drop in the number of CVEs assigned to lower-ranked CWEs. The top ten CWEs cover approximately 55.58% of the total number of CVEs, 57% of the total number of commits, and 55.75% of the total number of files.

We also investigated if it was possible to compute how long it took to fix the vulnerabilities, from the day that the CVE was disclosed to the project to the day that the fix was committed. Unfortunately, the NVD only includes the CVE *publication date*, which generally does not indicate when the vulnerability was discovered or when it was shared with the project. The actual disclosure date can be several months earlier if a responsible disclosure process is used. For example, Google's Project Zero delays publication for 90 days after disclosing vulnerabilities that are not actively being exploited, giving the option for earlier publication with mutual agreement. Other security researchers may use other time frames

**Table 4: Average number of days between CVE publication and vulnerability fix for all CVEs in *CVEfixes*.**

| mean | std | min | 25% | 50% | 75% | max |
|---|---|---|---|---|---|---|
| 171.55 | 414.93 | 0 | 7.0 | 28.0 | 123.25 | 7,636 |

**Table 5: Average number of days between CVE publication and vulnerability fix for the top ten projects with most CVEs (cf. Table 2a).**

| project | # CVEs | # CWEs | min | max | median | mean |
|---|---|---|---|---|---|---|
| linux | 973 | 53 | 0 | 4,582 | 49 | 213.64 |
| ImageMagick | 157 | 27 | 1 | 1,474 | 75 | 190.14 |
| tensorflow | 143 | 24 | 2 | 810 | 16 | 22.15 |
| tcpdump | 89 | 4 | 1 | 918 | 1 | 12.96 |
| FFmpeg | 83 | 17 | 1 | 1,523 | 44 | 63.22 |
| phpmyadmin | 67 | 13 | 1 | 1,211 | 13 | 36.15 |
| php-src | 60 | 17 | 3 | 855 | 27 | 50.26 |
| MISP | 50 | 17 | 0 | 430 | 1 | 29.33 |
| WordPress | 46 | 14 | 2 | 217 | 4 | 18.17 |
| hhvm | 40 | 18 | 1 | 1,328 | 37 | 182.18 |

or agree on them on a case-by-case basis with the projects. As a result of the variability this introduces in the time between disclosure and publication time, it is impossible to make strong statements on how long the projects really used to address the vulnerability.

With the above caveat in mind, Table 4 shows some statistics for the complete *CVEfixes* dataset on how many days it took from the CVE publication to committing a fix for the vulnerability. Moreover, Table 5 presents such statistics for the top ten projects with most CVEs, as earlier presented in Table 2a.

The last two characteristics that we will investigate are the vulnerability severity scores of CVEs in *CVEfixes*, as well as the impact that their fixes had on the maintainability of the project by means of the delta maintainability model (DMM). The delta maintainability model (DMM) [36] is a measure to compare and rank fine-grained code changes into low and high-risk changes. The overall DMM score refers to the proportion of low-risk changes in a commit, and is computed as the mean of three individual metrics that respectively measure the risk associated with code size, (cyclomatic) complexity, and size of the interface. The values of each of the DMM metrics range from 0 to 1, with higher values indicating better maintainability, i.e., lower risk changes, and low values indicating poor maintainability, i.e., high risk changes.

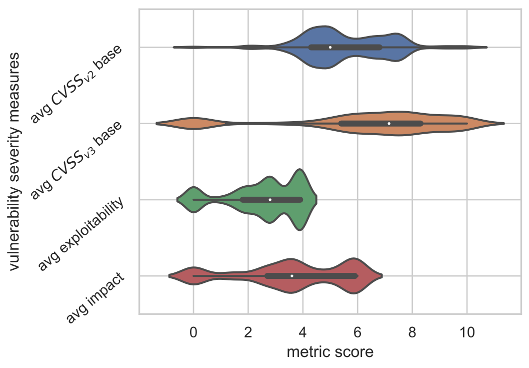

**Figure 3: Violin plot showing the distribution of average vulnerability severity scores for projects included in *CVEfixes***

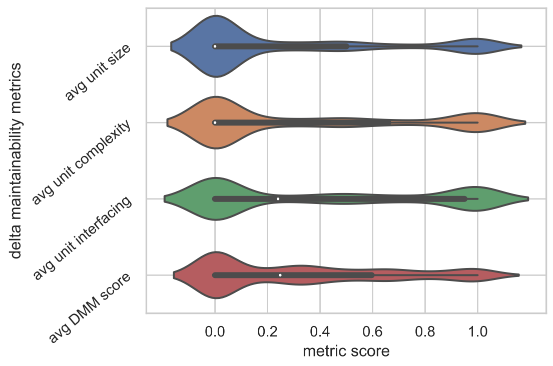

**Figure 4: Violin plot showing the distribution of average DMM scores for fixes to the projects included in *CVEfixes***

Figure 3 presents violin plots that show the distribution of CVSS$_{v2}$ and CVSS$_{v3}$ base scores, as well as exploitability and impact, for all vulnerabilities in *CVEfixes*, aggregated into averages per project. The width of each violin corresponds to the frequency of data points with that average. Inside the violin there is small box plot showing the ends of the first and third quartiles, and the median is indicated by the white dot. The values of these vulnerability severity metrics can vary between 0 and 10, with lower being better, and higher being worse. The figure suggests that the majority of the vulnerabilities have severity scores that are on the high side of the severity range, although their exploitability and impact tends to be on the lower sides of their respective ranges.

Figure 4 presents violin plots that show the distribution of DMM metrics of all vulnerability fixing commits in *CVEfixes*, again aggregated into averages per project. These metrics show how the maintainability of the projects was affected by the security fixes. Observe that the individual DMM metrics *unit size*, *unit complexity*, and *unit interfacing* mostly have a high density around 0 (indicating high risk changes with a detrimental effect on maintainability), and a smaller number of changes are considered low risk (with scores closer to 1). The overall *DMM score* is an aggregate computed as the mean of the individual metrics [36]. As such, it is more universally spread across different risk values, but still with a considerable number of commits requiring high-risk code change to fix the vulnerabilities, with the median around 0.25 and the third quartile around 0.6. An interesting question for follow-up research is investigating if vulnerability fixing commits that lower maintainability are followed by refactoring commits that restore maintainability.

## 5 DISCUSSION

### 5.1 Applications of *CVEfixes*

The *CVEfixes* dataset can be used in several ways to support data-driven software security research, for example, for automated vulnerability prediction, automated classification of identified vulnerabilities in Common Weakness Enumeration (CWE) types, vulnerability severity prediction, and automated repair of security vulnerabilities. The remainder of this section discusses several of these applications in more detail.

**Automated Vulnerability Prediction/Identification** The *CVEfixes* dataset contains different levels of vulnerability data, such as CVE-ID, version, description, type, publication date, current status, all interlinked up to the actual code changes that were introduced to fix the vulnerability. This data can be used to extract code features and metrics that help to better understand how security vulnerabilities are introduced in code. Moreover, the features and metrics can be used to model, train, and test vulnerability predictors based on classical machine learning approaches, and the textual descriptions, ranging from the CVE vulnerability level to the code level, can be used to train deep learning-based models for vulnerability prediction. Many studies have already targeted automated vulnerability identification using various machine learning models [7, 38–40], and Section 2 surveyed some of the challenges with the datasets used in these studies, such as limited size, lack of representativeness, and dataset imbalance. A comprehensive dataset like *CVEfixes* helps to overcome these challenges and enables a more thorough evaluation of the approaches.

**Automated Vulnerability Classification** The inclusion of vulnerability classifications in *CVEfixes* allows us to go one step further than automated vulnerability prediction, it also enables research on automated vulnerability classification, i.e., not just predicting the presence of a vulnerability but characterizing the type of the vulnerability. Such a classification is of interest since (automated) program repair approaches may have different efficacy and efficiency for different vulnerability types, so knowing the vulnerability type helps to inform which repair strategy to take. To address this challenge, $\mu$VulDeePeaker [4] and ManySStuBs4J [37] have constructed the multi-class vulnerability dataset. However, those datasets are specific to a given programming language and certain vulnerability/bug types. *CVEfixes*, on the other hand, uses the well-known CWE taxonomy to classify vulnerabilities in a hierarchy of vulnerability types and categories, and covers 27 programming languages (though 9 of these have fewer than 100 files changed in the commits covered by the dataset).

**Analysis of vulnerability fixing patches** Similar to how the vulnerable code in *CVEfixes* can be used to better understand how security vulnerabilities are introduced in code and how these can be automatically predicted, the fixes offered by *CVEfixes* can be used to analyze and build on vulnerability fixing patches. Several studies have already initiated research analyzing such patches, such as the detection of patterns that can be used in automated program repair [41], and the identification of security-relevant commits, also known as *pre-patches*, as these may inadvertently leak information about security vulnerabilities before the CVE is published [42, 43]. Other research has analyzed vulnerability fixing patches to facilitate the automated transformation of patches into "*honeypots*" that help trap malicious actors and detect if the corresponding vulnerabilities are exploited in the wild [44].

**Automated program repair** The pairs of vulnerable and fixed code provided by *CVEfixes* can be used to train machine learning-based automated program repair along the lines of SequenceR [45] and related work surveyed in Section 2, though we need to warn for a caveat w.r.t. the completeness of the patches, as discussed in more detail in the next section. Moreover, the current state of the art in this area is constrained with respect to the length of repairs that can be learned/synthesized, with 50 tokens being mentioned as an upper limit for acceptable performance [46]. By querying into *CVEfixes* data as shown in the previous subsection (and the example Jupyter Notebook in the distribution), *CVEfixes* facilitates the extraction of specific fixes that used only single (or a few) modified lines for fixing a vulnerability. This can be used to focus attention of the training process on cases with a lower number of modified tokens which are within reach of the technology, in order to help improve patch quality. Several studies [47–50] have already initiated work on the prediction of bugs/errors taking single line modified statements into account using the ManySStuBs4J dataset [37]. Querying *CVEfixes* in the way we presented earlier enables the extension of this work towards different programming languages, and vulnerability types.

### 5.2 Limitations and Future Extensions

One of the limitations of the current *CVEfixes* collection tool is that it collects all available fixes from scratch. We opted for this approach since any of the CVEs in the NVD may receive updates, so all should be inspected anyway. Nevertheless, it may be interesting to explore a more incremental update scheme that only clones a repository if not all of its fixes are already captured in the database, and augments the existing fixes in the other case. Such an approach would help to speed up the process of updating the database. One caveat is that we may also need to remove fixes from the database if they are no longer referenced by a CVE, but as we have seen, those fixes may be referred to by other CVEs as well, so some additional checks would be needed.

Another challenge/limitation that we have noticed is that some of the project repositories referenced in the NVD are no longer available. There can be multiple reasons; for example, the owners might have removed/changed/renamed their repositories, or they may have moved the repository between forges. This makes it impossible to fetch the fixes for these repositories. The current implementation tries to gather code from as many repositories as possible, and removes references to those that are unavailable to present an internally consistent view. However, this choice also means that we only present a subset of the information that is available in the NVD.

A last limitation is that a commit that is referenced as a vulnerability fixing commit in the NVD can still leave (part of) the vulnerability in the source code, and there may be later commits that complete the fix of the vulnerability, or the commit may contain changes unrelated to the fix [51]. To further improve the quality of the data, it would be of interest for future work to analyze consecutive patches, and select/untangle the code that together addresses the vulnerability.

Another direction for future work is upgrading our extraction tool to support mining the repositories from other issue-tracking and version control systems, i.e., Bugzilla, Mercurial, Subversion, etc. This will make it possible to gather vulnerability data from an even larger collection of projects.

Finally, considering the current controversy around ML models that are trained on (A)GPL licensed code and may possibly regurgitate some of that code as part of their operation, we plan to extend our datamodel with license information for the fixes included, so that users of *CVEfixes* can make an informed choice about including or excluding certain fixes from their training data.

## 6 CONCLUDING REMARKS

In this study, we propose the *CVEfixes* dataset and a fully automated collection tool that fetches this vulnerability dataset from publicly available information in the NVD and different version control systems. The dataset consists of real-world samples of vulnerable and corresponding patched code, their security metrics at five levels of abstraction, all linked to the CVE records. The collected information is enriched with various security and code related metrics and organized into a relational database to support easy querying.

The initial release of the dataset covers 5495 vulnerability fixing commits from 1754 open-source projects. This multi-level dataset establishes both qualitative and quantitative opportunities for vulnerability-related investigations. Researchers can study the relation from published CVEs all the way down to the corresponding code level vulnerabilities and their proposed fixes. The *CVEfixes* dataset enables research on vulnerability detection and classification, vulnerability severity prediction, (pre-)patch analysis, automated vulnerability repair, and many others. We plan to periodically update *CVEfixes*, and extend it with mining other open-source projects from different version control systems and issue tracking systems.

## ACKNOWLEDGMENTS

The work presented in this paper has been financially supported by the Research Council of Norway through the secureIT project (RCN contract #288787).

## REFERENCES

[1] MITRE. *Common Vulnerabilities and Exposures (CVE).* URL: https://cve.mitre.org/ (visited on May 28, 2021).
[2] M.-j. Choi, S. Jeong, H. Oh, and J. Choo. “End-to-End Prediction of Buffer Overruns from Raw Source Code via Neural Memory Networks.” In: *Int'l Joint Conf. Artificial Intelligence.* 2017, pp. 1546–1553.
[3] P. Morrison, K. Herzig, B. Murphy, and L. Williams. “Challenges with Applying Vulnerability Prediction Models.” In: *Symp. and Bootcamp on the Science of Security.* ACM, 2015, pp. 1–9.
[4] D. Zou, S. Wang, S. Xu, Z. Li, and H. Jin. “$\mu$VulDeePecker: A Deep Learning-Based System for Multiclass Vulnerability Detection.” In: *IEEE Trans. Dependable and Secure Computing* (2019), p. 13.
[5] X. Wang, K. Sun, A. Batcheller, and S. Jajodia. “Detecting "0-Day" Vulnerability: An Empirical Study of Secret Security Patch in OSS.” In: *Int'l Conf. Dependable Systems and Networks.* 2019, pp. 485–492.
[6] Z. Feng et al. “CodeBERT: A Pre-Trained Model for Programming and Natural Languages.” In: *Findings of the Association for Computational Linguistics: EMNLP 2020.* 2020, pp. 1536–1547.
[7] R. Russell, L. Kim, L. Hamilton, T. Lazovich, J. Harer, O. Ozdemir, P. Ellingwood, and M. McConley. “Automated Vulnerability Detection in Source Code Using Deep Representation Learning.” In: *Int'l Conf. Machine Learning and Applic.* IEEE, 2018, pp. 757–762.
[8] R. Coulter, Q.-L. Han, L. Pan, J. Zhang, and Y. Xiang. “Code Analysis for Intelligent Cyber Systems: A Data-Driven Approach.” In: *Information Sciences* 524 (2020), pp. 46–58.
[9] G. Lin, S. Wen, Q.-L. Han, J. Zhang, and Y. Xiang. “Software Vulnerability Detection Using Deep Neural Networks: A Survey.” In: *Proceedings of the IEEE* (2020), pp. 1–24.
[10] T. Boland and P. E. Black. “Juliet 1.1 C/C++ and Java Test Suite.” In: *IEEE Computer* 45.10 (2012), pp. 88–90.
[11] P. E. Black. “A Software Assurance Reference Dataset: Thousands of Programs With Known Bugs.” In: 123 (2018).

[12] M. Gupta, M. Govil, and G. Singh. “Text-Mining and Pattern-Matching Based Prediction Models for Detecting Vulnerable Files in Web Applications.” In: *J. Web Eng.* 17.1-2 (2018), pp. 28–44.

[13] M. K. Gupta, M. C. Govil, and G. Singh. “Predicting Cross-Site Scripting (XSS) Security Vulnerabilities in Web Applications.” In: *Int'l Joint Conf. Computer Science and Softw. Eng.* 2015, pp. 162–167.

[14] S. A. Mokhov, J. Paquet, and M. Debbabi. “MARFCAT: Fast Code Analysis for Defects and Vulnerabilities.” In: *Int'l Ws. Softw. Analytics.* 2015, pp. 35–38.

[15] I. Medeiros, N. F. Neves, and M. Correia. “Automatic Detection and Correction of Web Application Vulnerabilities Using Data Mining to Predict False Positives.” In: *Int'l Conf. World Wide Web.* ACM, 2014, pp. 63–74.

[16] S. Mokhov, J. Paquet, and M. Debbabi. “The Use of NLP Techniques in Static Code Analysis to Detect Weaknesses and Vulnerabilities.” In: *Canadian Conf. Artificial Intelligence.* Vol. 8436. 2014, pp. 326–332.

[17] Y. Zheng, S. Pujar, B. Lewis, L. Buratti, E. Epstein, B. Yang, J. Laredo, A. Morari, and Z. Su. “D2A: A Dataset Built for AI-Based Vulnerability Detection Methods Using Differential Analysis.” In: *Int'l Conf. Softw. Eng.: Softw. Eng. in Practice.* IEEE, 2021, pp. 111–120.

[18] D. Mitropoulos, G. Gousios, P. Papadopoulos, V. Karakoidas, P. Louridas, and D. Spinellis. “The Vulnerability Dataset of a Large Software Ecosystem.” In: *Int'l Ws. Building Analysis Datasets and Gathering Experience Returns for Security.* IEEE, 2014, pp. 69–74.

[19] *FindBugs.* URL: http://findbugs.sourceforge.net/ (visited on Jan. 25, 2021).

[20] R. Raducu, G. Esteban, F. J. Rodríguez Lera, and C. Fernández. “Collecting Vulnerable Source Code from Open-Source Repositories for Dataset Generation.” In: *Applied Sciences* 10.4 (2020), p. 1270.

[21] Y. Zhou, S. Liu, J. Siow, X. Du, and Y. Liu. “Devign: Effective Vulnerability Identification by Learning Comprehensive Program Semantics via Graph Neural Networks.” In: *Conf. Neural Information Processing Systems.* 2018, p. 11.

[22] Z. Li, D. Zou, S. Xu, X. Ou, H. Jin, S. Wang, Z. Deng, and Y. Zhong. “VulDeePecker: A Deep Learning-Based System for Vulnerability Detection.” In: *Network and Distributed System Security Symp.* 2018.

[23] R. Vieira, A. da Silva, L. Rocha, and J. P. Gomes. “From Reports to Bug-Fix Commits: A 10 Years Dataset of Bug-Fixing Activity from 55 Apache's Open Source Projects.” In: *Int'l Conf. Predictive Models and Data Analytics in Softw. Eng.* ACM, 2019, pp. 80–89.

[24] H. Alves, B. Fonseca, and N. Antunes. “Software Metrics and Security Vulnerabilities: Dataset and Exploratory Study.” In: *European Dependable Computing Conf.* 2016, pp. 37–44.

[25] A. Gkortzis, D. Mitropoulos, and D. Spinellis. “VulinOSS: A Dataset of Security Vulnerabilities in Open-Source Systems.” In: *Int'l Conf. Mining Softw. Repositories.* ACM, 2018, pp. 18–21.

[26] M. Jimenez, Y. Le Traon, and M. Papadakis. “Enabling the Continuous Analysis of Security Vulnerabilities with VulData7.” In: *Int'l Working Conf. Source Code Analysis and Manipulation.* 2018, pp. 56–61.

[27] 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: *Int'l Conf. Mining Softw. Repositories.* 2019, pp. 383–387.

[28] J. Fan, L. Li, S. Wang, and T. N. Nguyen. “A C/C++ Code Vulnerability Dataset with Code Changes and CVE Summaries.” In: *Int'l Conf. Mining Softw. Repositories.* 2020, p. 5.

[29] G. Lin, W. Xiao, J. Zhang, and Y. Xiang. “Deep Learning-Based Vulnerable Function Detection: A Benchmark.” In: *Int'l Conf. Information and Communications Security.* Vol. 11999. 2020, pp. 219–232.

[30] Y. Ma, T. Dey, C. Bogart, S. Amreen, M. Valiev, A. Tutko, D. Kennard, R. Zaretzki, and A. Mockus. “World of Code: Enabling a Research Workflow for Mining and Analyzing the Universe of Open Source VCS Data.” In: *Empirical Softw. Eng.* 26.2 (2021), p. 22.

[31] S. Dueñas, V. Cosentino, G. Robles, and J. M. Gonzalez-Barahona. “Perceval: Software Project Data at Your Will.” In: *Int'l Conf. Softw. Eng.* ACM, 2018, pp. 1–4.

[32] D. Spadini, M. Aniche, and A. Bacchelli. “PyDriller: Python Framework for Mining Software Repositories.” In: *Joint European Softw. Eng. Conf. and Symp. Found. Softw. Eng.* ACM, 2018, pp. 908–911.

[33] NIST. *National Vulnerability Database (NVD).* URL: https://nvd.nist.gov/ (visited on July 22, 2020).

[34] MITRE. *Common Weakness Enumeration (CWE).* URL: https://cwe.mitre.org/ (visited on May 28, 2021).

[35] Forum of Incident Response and Security Teams (FIRST). *Common Vulnerability Scoring System (CVSS).* URL: https://www.first.org/cvss (visited on May 28, 2021).

[36] M. di Biase, A. Rastogi, M. Bruntink, and A. van Deursen. “The Delta Maintainability Model: Measuring Maintainability of Fine-Grained Code Changes.” In: *Int'l Conf. Technical Debt.* IEEE, 2019, pp. 113–122.

[37] R.-M. Karampatsis and C. Sutton. “How Often Do Single-Statement Bugs Occur? The ManySStuBs4J Dataset.” In: *Int'l Conf. Mining Softw. Repositories.* 2020, pp. 573–577.

[38] X. Li, L. Wang, Y. Xin, Y. Yang, and Y. Chen. “Automated Vulnerability Detection in Source Code Using Minimum Intermediate Representation Learning.” In: *Applied Sciences* 10.5 (2020).

[39] N. Saccente, J. Dehlinger, L. Deng, S. Chakraborty, and Y. Xiong. “Project Achilles: A Prototype Tool for Static Method-Level Vulnerability Detection of Java Source Code Using a Recurrent Neural Network.” In: *Int'l Conf. Autom. Softw. Eng.* IEEE, 2019, pp. 114–121.

[40] H. K. Dam, T. Tran, T. T. M. Pham, S. W. Ng, J. Grundy, and A. Ghose. “Automatic Feature Learning for Predicting Vulnerable Software Components.” In: *IEEE Trans. Softw. Eng.* 47.1 (2018), pp. 67–85.

[41] P. Hegedűs. “Inspecting JavaScript Vulnerability Mitigation Patches with Automated Fix Generation in Mind.” In: *Int'l Conf. Computational Science and Its Applic.* Vol. 12252 LNCS. 2020, pp. 975–988.

[42] M. Yang, J. Wu, S. Ji, T. Luo, and Y. Wu. “Pre-Patch: Find Hidden Threats in Open Software Based on Machine Learning Method.” In: *World Congress on Services.* Vol. 10975. 2018, pp. 48–65.

[43] A. Sabetta and M. Bezzi. “A Practical Approach to the Automatic Classification of Security-Relevant Commits.” In: *Int'l Conf. Softw. Maintenance and Evolution.* IEEE, 2018.

[44] A. Larmuseau and D. Shila. “PatchSweetner: Exploit Detection Through the Automatic Transformation of Security Patches.” In: *Military Communications Conf.* 2018, pp. 939–945.

[45] Z. Chen, S. J. Kommrusch, M. Tufano, L.-N. Pouchet, D. Poshyvanyk, and M. Monperrus. “SEQUENCER: Sequence-to-Sequence Learning for End-to-End Program Repair.” In: *IEEE Trans. Softw. Eng.* (2019).

[46] Z. Chen, S. Kommrusch, and M. Monperrus. *Using Sequence-to-Sequence Learning for Repairing C Vulnerabilities.* arXiv: 1912.02015. CoRR e-print, 2019.

[47] E. Mashhadi and H. Hemmati. “Applying CodeBERT for Automated Program Repair of Java Simple Bugs.” In: *Int'l Conf. Mining Softw. Repositories.* IEEE, 2021, p. 5.

[48] B. Mosolygó, N. Vándor, G. Antal, and P. Hegedűs. “On the Rise and Fall of Simple Stupid Bugs: A Life-Cycle Analysis of SStuBs.” In: *Int'l Conf. Mining Softw. Repositories.* IEEE, 2021, p. 5.

[49] F. Madeiral and T. Durieux. “A Large-Scale Study on Human-Cloned Changes for Automated Program Repair.” In: *Int'l Conf. Mining Softw. Repositories.* IEEE, 2021, p. 5.

[50] J. Hua and H. Wang. “On the Effectiveness of Deep Vulnerability Detectors to Simple Stupid Bug Detection.” In: *Int'l Conf. Mining Softw. Repositories.* IEEE, 2021, p. 5.

[51] K. Herzig, S. Just, and A. Zeller. “The Impact of Tangled Code Changes on Defect Prediction Models.” In: *Empirical Softw. Eng.* 21.2 (2016), pp. 303–336.