A Vulnerability Introducing Commit Dataset for Java: An Improved SZZ based Approach

Tamás Aladics[1,2][a], Péter Hegedűs[1,2][b] and Rudolf Ferenc[1][c]

[1]*Department of Sofware Engineering, University of Szeged, Szeged, Hungary*

[2]*FrontEndART Ltd., Szeged, Hungary*

Keywords: Just-in-Time Vulnerability Detection, Dataset, SZZ, Vulnerability Introducing Commits.

Abstract: In the domain of vulnerability detection from the source code by applying static analysis, the number and quality of available datasets for creating and testing security analysis methods is quite low. To be precise, there are already several public datasets containing vulnerability fixing commits; however, vulnerability introducing commit datasets are scarce, which would be essential for creating and validating just-in-time vulnerability detection approaches. In this paper, we propose an SZZ (an algorithm originally developed to find bug introducing commits) based method with a specific filtering mechanism to create vulnerability introducing commit datasets from vulnerability fixes. The filtering phase involves measuring a relevance score for each vulnerability introducing commit candidates based on commit similarities. We generated a novel Java vulnerability introducing dataset from the existing project-KB repository to demonstrate our algorithm's capabilities. We also showcase the generated database and the effectiveness of our filtering method through several hand-picked examples from the dataset.

1 INTRODUCTION

Many software engineering-related tasks, such as quality assurance or testing, are now aided by machine learning, which relies heavily on the abundance of data. Most of these tasks are typically based on machine learning, therefore the availability of datasets is crucial to train reliably and to get a generally well-performing model.

Fortunately, when the goal is related to vulnerability fixes, there are already well established datasets that can be relied on. These datasets typically contain validated code changes (i.e. commits) that fix a particular vulnerability described in a Common Vulnerabilities and Exposures (CVE) (MITRE Corporation, v 21) entry, a publicly disclosed security vulnerability in a software system. One such dataset is published as part of the repository "project-KB" (project kaybee) (Ponta et al., 2019) maintained by SAP.

This dataset contains CVE entries and their corresponding commit references of Java software systems that are known to have fixed the security issues. Datasets like project-KB can be exploited in various use cases such as aggregating security related statistics, getting insight on the security state of a system and also checking which version of a specific library or software contains a security risk. The latter use case is crucial for larger projects that use many other (typically open source) software components to know what kind of security issues is the system is exposed to by using those external libraries.

There are tasks, however, that are not related to vulnerability fixes but to vulnerability introducing commits, such as just-in-time vulnerability detection and localization (Amin et al., 2019; Cao et al., 2021; Li et al., 2013), when the purpose is to find the vulnerable part of the system or detect the presence of a security bug. In these cases, finding the appropriate dataset can be challenging, if possible at all. This is partly due to the fact that while fixing commits are sometimes available as part of the CVE entries (or at least it is possible to examine the commit history and apply heuristics to identify the fixing commit (Ponta et al., 2019)), the commit which introduced that particular vulnerability can be ambiguous and not trivial to find.

Our aim in this work is to help solve the lack of vulnerability introducing commit datasets by provid-

[a] https://orcid.org/0000-0002-4689-8878
[b] https://orcid.org/0000-0003-4592-6504
[c] https://orcid.org/0000-0001-8897-7403

Aladics, T., Hegedűs, P. and Ferenc, R.
A Vulnerability Introducing Commit Dataset for Java: An Improved SZZ based Approach.
DOI: 10.5220/0011275200003266
In *Proceedings of the 17th International Conference on Software Technologies (ICSOFT 2022)*, pages 68-78
ISBN: 978-989-758-588-3; ISSN: 2184-2833

ing a method to automatically generate them from vulnerability fixing datasets. The procedure consists of two phases, the first involves running an implementation of the SZZ algorithm (Sliwerski et al., 2005) called SZZ Unleashed (Borg et al., 2019) for each fixing commit in the dataset, which results in a set of candidate introducing commits for each fixing commit. However, inspecting these candidate commits shows that the results from the first phase are hardly usable in practice due to a number of issues. The most prominent of these issues are that the number of the candidates and more importantly the number of false positives can be very high.

Therefore, we designed a second phase, which involves the filtering of the results produced by the first phase to overcome the issues. The filtering involves selecting the top *n* most relevant commits, where *n* is an arbitrarily chosen number, and commit relevance is determined by a metric called the *relevance score*. The relevance score is a heuristic that is assigned to each candidate introducing commit and it measures the commit's vulnerability introducing relevance: the higher it is the more likely that the commit is indeed a real introducing commit. The calculation of the relevance score and the filtering process is discussed in more details in Section 4.

Applying the method briefly explained, we generated a novel Java vulnerability introducing commit dataset from the project-KB vulnerability fixing dataset and made it publicly available[1] together with the tools implementing the generation process. To summarize, the main contributions of our work are as follows:

- We propose a two-phase method to automatically generate accurate vulnerability introducing commit datasets from vulnerability fixes;
- As part of our method, we suggest a way to measure introducing commit relevance to a fixing commit, which we refer to as the *relevance score*;
- We provide a toolchain that can be used to generate new vulnerability introducing datasets from existing repositories, similar to the project-KB dataset;
- We publish a novel Java vulnerability introducing commit dataset created from the project-KB repository using our proposed method and tool.

The rest of the paper is organized as follows. We list the works related to ours in Section 2. Section 3 gives a motivation for our research through a running example. We describe the technical details of the proposed two-phase method for generating vulnerability introducing commit datasets in Section 4. We demonstrate the usage of the proposed method by creating a novel Java vulnerability introducing commit dataset, which is presented in Section 5. In Section 6, we enumerate the possible threats to the validity of our work, while we conclude the paper in Section 7.

[1]https://doi.org/10.5281/zenodo.5785239

2 RELATED WORK

Lately, the number of vulnerabilities is increasing at an alarming rate, which is mainly traceable by the disclosed open source software vulnerability entries. According to the report by WhiteSource (whi, c 14), the number of published open source software vulnerabilities in 2020 rose by over 50% compared to the previous year, from 6111 to 9658. This sharp increase is an unambiguous indicator of the ever-growing problem of software vulnerabilities and also shows the urgent need to understand them better and faster.

To understand security issues, analyze them, draw conclusions, or build tools that can help managing these issues, systematically gathered collections of data are essential. There are a couple of datasets (Gkortzis et al., 2018; Ponta et al., 2019) available containing information about vulnerability fixes (i.e. set of commits fixing a known vulnerability and the source code version before and after this fix). Most of them build on the information contained in the Common Vulnerabilities and Exposures (CVE) repository (cve, v 20) of publicly disclosed vulnerabilities. CVE provides detailed information about the specific vulnerability, in particular a unique identifier (CVE-ID), a description and a set of public references. The National Vulnerability Database or NVD (nvd, v 20) contains practically all vulnerabilities in CVE (except some that are pending at the time but will be added later) and extends them with additional information such as vulnerability type (CWE) and severity scores (CVSS).

Vulnerability fixing datasets leverage the information present in the CVE and NVD databases, which sometimes contain links to the actual fixing patches of a vulnerability. The project-KB dataset (pro, c 14) is part of the project-KB repository maintained by SAP. It contains manually curated entries of vulnerability fixes in Java projects, where most of these entries have a corresponding CVE record. The authors describe this dataset and publish a snapshot of it in a separate publication (Ponta et al., 2019).

Another dataset that involves automatic collection of CVE entries from NVD is VulinOSS (Gkortzis et al., 2018), which contains reported vulnerabilities of 8694 open-source project versions. As part of their research, the authors supplemented the corresponding source code with various source code metrics.

Yunhui Zhengi et al. (Zheng et al., 2021) use static analyzer tools to generate a fixing commit dataset specifically for machine learning uses. First, they collect several candidate fixing commits using machine learning methods then they use differential analysis: they run static analysis on the before and after commit versions of the fixing commit. If a set of issues detected before the fixing commit disappear in the after state, they label it as positive, otherwise the fixing commit is labeled as negative.

Guru Prasad Bhandari et al. (Bhandari et al., 2021) as the main contribution of their research published the tool CVEFixes, which can automatically generate a fixing commit database by parsing and validating every record from NVD currently available. The initial release in 2021 contained all published CVEs up to 9 June, covering 5365 CVE records.

Datasets of vulnerability fixing commits can be used for a wide variety of downstream tasks, such as locating security patches (Tan et al., 2021; Li and Paxson, 2017; Wang et al., 2021b; Wang et al., 2021a), vulnerable code clone detection (Woo et al., 2021; Xiao et al., 2020), and patch presence testing (Dai et al., 2020; Falleri et al., 2014). Tan et al. (Tan et al., 2021) facilitate security patch detection by vulnerability commit correlation ranking. They ranked commits by training a RankNet model on features they parsed from commits and CVE vulnerability entries. Our research has a similar goal but while they focused on fixing commits, we target vulnerability introducing commits. Additionally, they used machine learning models to achieve the ranking, while in our work we follow a simpler approach.

As it can be seen, there are various datasets available when the task is related to vulnerability fixes. In the case of vulnerability introducing commits, however, the available resources are a lot more scarce. Meneely et al. and Shin et al. investigate source code repository metadata in relation with CVE entries (Meneely and Williams, 2012; Meneely et al., 2013; Meneely et al., 2014). Using features like code churn and lines of code they created a database from mappings of CVEs to commits for the Mozilla Firefox Browser, Apache HTTP server and parts of the RHEL Linux kernel. However, this database is not publicly accessible and also not scalable, since it is manually constructed.

One attempt to automatize this process is an approach called *VCCFinder* by Henning Perl et al. (Perl et al., 2015). In their work, the authors describe a mapping of CVEs to GitHub commits in order to create a *vulnerability contributing commit (VCC)* database. This mapping is based on a heuristic that involves the *git blame* command and some filtering, such as excluding lines in documentation. This work has a similar goal to our research, even though we took a different approach at some points.

In contrast to VCCFinder, in our work we used an enhanced version of the well-known SZZ algorithm (Sliwerski et al., 2005), called SZZ Unleashed (Borg et al., 2019) to find the introducing commits. For a commit, that is said to be bug introducing, the SZZ algorithm is using the *git blame* command (which maps each line in the commit to the last modifier) to find all of the commits that directly preceded it. After that, additional steps are made to filter out non bug-related commits by using various information, such as the bug report date. SZZUnleashed provides various improvements over the base SZZ algorithm detailed in their work, like line-mappings and the support for git based issue trackers. As opposed to VCCFinder, SZZ considers more information and it will produce a set of candidate vulnerability introducing commits, while VCCFinder produces at most one (the one with most lines blamed). This gives us the possibility to identify multiple commits as introducing, which is the case in many real-world problems (a vulnerability can be introduced through multiple commits). We also provide more flexibility on the introducing commit filtering phase, such as choosing file extension, and we also propose a way to measure the relevance of each candidate commit as well as each related commit file's contribution score. This way, the user can gain insight into the ranking process and adjust it accordingly.

3 OVERVIEW AND MOTIVATION

In this section we demonstrate the motivation behind our research and give intuition on how our method works through a running example. We discuss our method in more detail in Section 4.

As already mentioned, the starting point of our approach is having a vulnerability fixing commit (VFC) for which we want to generate a set of introducing commits (VIC). VFCs can be found in VFC datasets such as project-KB (Ponta et al., 2019), and in most of the cases a VFC can be linked to a corresponding CVE id (i.e. to the actual vulnerability it fixes). One such VFC is linked to the *CVE 2016-3674* (cve, c 14a) entry, a vulnerability allowing an attacker to perform an XML external entity attack (Herzog, 2010) in multiple components of the XStream (xst, c 14) project, a Java to XML serializer library. This vulnerability occurs in multiple files, such as `Dom4JDriver`, `DomDriver`, `SjsxpDriver`, `StaxDriver`, and 3 more.

```
commit sha: c9b121a88664988ccbabd83fa27bfc2a5e0bd139
++- xstream/src/.../io/xml/StaxDriver.java


//Before applying fix
protected XMLInputFactory createInputFactory() {
  return XMLInputFactory.newInstance();
}
```

```
//After applying fix
protected XMLInputFactory createInputFactory() {
  final XMLInputFactory instance = XMLInputFactory.
        newInstance();

  instance.setProperty(XMLInputFactory.
        IS_SUPPORTING_EXTERNAL_ENTITIES, false);
  return instance;
}
```

Figure 1: Before and after applying the changes in file StaxDriver.java in project XStream as part of fixing CVE-2016-3674.

Figure 1 shows the affected source code state before and after applying the fix in the vulnerable `StaxDriver .java` file (only the relevant changed source code is shown). It can be observed that the commit fixing this vulnerability simply sets an appropriate flag on the XMLInputFactory. Our goal is to find the commit that introduced changes that led to this vulnerability, that is, yield the *"Before applying fix"* state in Figure 1. In this particular example, the commit that added the XMLInputFactory instantiation statement without setting the `IS_SUPPORTING_EXTERNAL_ENTITIES` flag to *false*.

To achieve this, we choose to use an open-source implementation of a recent variant of the SZZ algorithm, called SZZ Unleashed (Borg et al., 2019) to find a set of possible introducing commits. SZZ (Sliwerski et al., 2005) was originally designed to provide a process to automatically identify the fix inducing lines to lines that are changed in a bug-fixing commit. Since the vulnerability occurs in multiple files, it is very likely that it has been introduced through multiple commits. We run SZZ Unleashed as part of our own proposed tool, called *BugIntroducerMiner* to generate the VICs. BugIntroducerMiner is a simple wrapper around SZZ Unleashed and its purpose is to run SZZ Unleashed on commits stored in VFC dataset, in our case on project-KB. In Figure 2, we can see the results of our tool for the VFC shown in Figure 1, which has the commit hash `c9b121...`[2]

As discussed before, the fix is rather simple and involves adding a line that sets a specific flag. However, due to the fact that the vulnerability occurs in multiple files, SZZ found 17 candidate introducing

```
CVE-2016-3674:
  commitsWithIntroducers:
    c9b121a88664988ccbabd83fa27bfc2a5e0bd139:
      [deec01beaa1bd878f7acda9f035a39238a217ae9,
      bba4bc28e62073f9baac9c58cbc14de958df3b7e,
      72efd4a37f0ab81d2dfeb013d35ec7cbed0510b1,
      ...
      1b0f802b01632954c6ba2a6605592e3e2975f72f,
      4fd39f2f2616d4ea9e1d25d30dc78931be01dfb0,
      c9794d2f905985c8e45fa4d77525c130a5fd0a20]
  repo: https://github.com/x-stream/xstream
```

Figure 2: Result of running the BugIntroducerMiner tool on the VFC corresponding to CVE 2016-3674.

commits. This is hardly manageable because a lot of practical uses prefer that for each CVE entry we have only few (ideally just one) introducing commit. To make things worse, manually checking the candidate commits we found that many of them are false positives or contribute little to the vulnerability introduction. For example, the changes are made in comments, or in source code next to the fix location (i.e in a neighboring row that SZZ still considers), or happened in files that the user is not interested in (configuration files instead of java source files).

For these reasons, we applied an additional filtering step, which we implemented as another tool, *FilterBugIntroducer*. The filtering is based on ranking the candidate commits according to their *relevance score* that is calculated by measuring similarity between the candidate commits and the source code state before the fixing commit. The calculated scores for our running example can be seen on Figure 3. We will detail how these scores are calculated in the following sections, but here we briefly summarize their purpose. For each candidate VIC we calculate a score called *relevance score* (*Overall score* in the figure) that measures how relevant that commit is as vulnerability introducing. This score is calculated by aggregating the *contribution scores* (*Total* in the figure) for each file corresponding to the VIC. Contribution score corresponds to the file's contribution to the vulnerability. It is calculated by multiplying the file's similarity to the fixing file (calculated as the portion of identical lines in all lines) with the fixing file's *base score*, where the base score is just a simple metric that denotes the quanitity of changes happened in that specific file relative to all the changes in the commit. In the figure, the *Overall score* denotes the relevance score we used to rank the candidate VICs.

For our running example, we choose the top 2 candidate VICs, which are displayed in Figure 3. The fix patch with the highest relevance score can be seen in Section 5 (Figure 8), where we further discuss the benefits of our results over plain SZZ. After manually inspecting these commits, we can conclude that the filtering produced reasonable results:

- `4fd39...` introduces the vulnerability in multiple

[2]In the figures, we show only part of the results to remain concise. Omitting data is marked with '...'

files, for example, file `SjsxpDriver.java` is created in this commit and the vulnerable part has not changed until the VFC. In file `StaxDriver.java`, the vulnerability is introduced in the method added in this commit (i.e. the method contains the instantiation without setting the appropriate flag).

- `72efd...` changed several files that are patched in the fix with multiple smaller changes whose result is changed in the VFC.

After this brief overview and motivating example, in the following sections we are elaborating on the way we are performing the mapping of VFCs to sets of VICs, we explain how we calculate the relevance scores and how can these results be used in general.

```
============ CVE−2016−3674 ============

Repo: https://github.com/x−stream/xstream
SHA: c9b121a88664988ccbabd83fa27bfc2a5e0bd139

File base scores:
  SjsxpDriver.java: 0.25982952983227936
  StandardStaxDriver.java: 0.3634863898817707
  StaxDriver.java: 0.2073137200989827
  WstxDriver.java: 0.16937036018696727

Introducing commit SHAs:
  ...
  − 4fd39f2f2616d4ea9e1d25d30dc78931be01dfb0
    − SjsxpDriver.java:
      Similarity: 0.7142857142857143
      Contribution: 0.18559252130877096
    − StaxDriver.java:
      Similarity: 0.4
      Contribution: 0.08292548803959308
    − WstxDriver.java:
      Similarity: 0.5714285714285714
      Contribution: 0.09678306296398129
  Relevance score: 0.3653010723123453
  ...

  − 72efd4a37f0ab81d2dfeb013d35ec7cbed0510b1
    − SjsxpDriver.java:
      Similarity: 0.2857142857142857
      Contribution: 0.07423700852350838
    − StandardStaxDriver.java:
      Similarity: 0.21428571428571427
      Contribution: 0.07788994068895086
    − StaxDriver.java:
      Similarity: 0.4
      Contribution: 0.08292548803959308
    − WstxDriver.java:
      Similarity: 0.21428571428571427
      Contribution: 0.036293648611492986
  Relevance score: 0.2713460858635453
```

Figure 3: The calculated relevance scores per candidate VIC (*Relevance score*), contribution scores for each file corresponding to a VIC (*Contribution*), and the base scores for each file in the VFC (*File base scores*).

4 METHODOLOGY

One of the main contributions of this paper is a two-phase method to generate VIC datasets from VFC databases. The two phases of the method are:

1. **Identifying the Vulnerability Introducing Commits (VICs):** Run SZZ Unleashed for each commit in a vulnerability fixing commit (VFC) database to identify a set of candidate VICs. We implemented a tool called *BugIntroducerMiner* that is able to perform this phase for databases structured like project-KB.
2. **Filtering:** Taking the previous phase's output (the SZZ results) as input, we perform a filtering phase (using another tool we created, called *FilterBug-Introducer*). The output of this phase is the top *n* most relevant commits ranked by their *relevance scores*, where *n* is an arbitrarily chosen number.

4.1 Phase 1 - Identifying the Introducing Commits

The input to our proposed VIC extraction algorithm is, as already discussed, a VFC database. We adjusted this algorithm to databases structured like the project-KB dataset, which we briefly described in Section2, however, the general idea discussed here can easily be applied to different datasets as well.

To understand the properties of a typical VFC database, we describe the structure of the project-KB dataset (i.e. the dataset we use to demonstrate our method), which can be seen in Figure 4a. The database contains its data organized into folders named after the CVE identifiers of the vulnerabilities to which fixing commits are published. Each folder contains a `statement.yaml` file that describes the found vulnerability fixing commits linked to the CVE. Figure 4b shows an example statement file for the vulnerability referenced as CVE-2008-1728. It can be seen that a vulnerability fixing entry contains some metadata, such as the textual description of the vulnerability, the CVE id and a section *fixes* that identifies VFCs, such as the repository URL, the branch and the commit hash.

The statement.yaml contain all the necessary information about the VFCs in the database. Our goal was to extract the VICs for each VFC entry and we made some decisions regarding the parsing:

- A statement.yaml file's *fixes* section can have multiple elements. This happens when a vulnerability is fixed in multiple branches or in different repositories. Usually, the master branch of the main repository is the first element of the *fixes* section, so we chose that as the fix. Other entries are typically the duplicates of the same fix in other branches.
- A statement.yaml file's *fix* (an element of the *fixes* section) can have multiple associated commits. This happens when a vulnerability fix involves multiple commits. In such cases, we choose the

a) Repository structure

```
<repo_root>/
  statements/
    CVE-2005-3745/
      statememt.yaml
    CVE-2006-1546/
      statement.yaml
      ...
  LICENSE.txt
  README.md
  ...
```

b) Example statement.yaml file

```
vulnerability_id: CVE-2008-1728
notes:
- text: ConnectionManagerImpl.java in Ignite Realtime
      Openfire 3.4.5 allows remote authenticated users to
      cause a denial of service (daemon outage) by
      triggering large outgoing queues without reading
      messages.
fixes:
- id: DEFAULT_BRANCH
  commits:
  - id: c9cd1e521673ef0cccb8795b78d3cbaefb8a576a
  repository: https://github.com/igniterealtime/Openfire
```

Figure 4: Project-KB structure (**a**) with an example statement file (**b**).

latest commit as it will contain all the previous changes, and as so it represents the final, ”fixed” state.

After parsing the database, we get a set of VFCs for which we try to identify VICs by running SZZ Unleashed, an implementation of the SZZ algorithm. An example output of this process can be seen in Figure 2 (the complete output of this phase contains multiple such CVE elements).

In summary, phase 1 involves parsing every vulnerability fixing entry in the source database to get a set of VFCs. Then, SZZ Unleashed is run on each VFC and the results from multiple SZZ Unleashed runs are aggregated to get a candidate VIC database.

However, as we mentioned in Section 3, the SZZ algorithm's results are hardly usable as is for a number of reasons:

- SZZ takes into account every change in the commit and it cannot be configured to detect changes only in a special file type. So it is possible that a change happens in a documentation file and the change will generate many false positive introducing commits.
- SZZ does not offer a ranking on the provided results, so if a fix is complex, involving multiple files and multiple changes per file, the extracted number of introducing commits are too large to handle, even if we exclude false positives. In such cases, a way to choose the commit with most relevance to the vulnerability would be welcome. For example, running SZZ Unleashed on CVE-2016-2141 results in a VIC set of 634 elements.
- Results are problematic to explain as no detailed information is provided of the way the candidate VICs were chosen, making it hard to draw conclusions from them.

Taking these issues into consideration, we designed a filtering phase, which aims to address the above mentioned problems.

4.2 Phase 2 - Filtering

The input for this phase is the output of the first phase, that is a list in which every element is a pair of VFC and a set of candidate VICs (like in Figure 2). Our aim in this phase is to provide a score for each candidate VIC that measures its relevance to the VFC. We refer to this score as *relevance score*, and it can be calculated for a pair of VFC and VIC. The score calculation algorithm is shown in Figure 5a as a pseudo code.

Relevance score is produced by iterating over each file changed in the candidate introducing commit and identifying and comparing it to the corresponding file in the fixing commit - if it exists. This usually involves an enhanced name check for equality that also considers name changes through the git history. In the pseudo code, the function `get_by_name` refers to this action and returns the corresponding fixing file or the `None` value if it is not found. If the value is not `None`, it is possible to continue with calculating the contribution score as the product of the fixing file's base score and the similarity between the fixing file and the candidate introducing commit file. The final relevance score is simply the sum of the contribution scores calculated for all pairs of changed files in the fixing and introducing commits.

a) Relevance score

```
relevance_score = 0
for introducing_file in introducing_commit.files:
  fixing_file = fixing_commit.fixing_files.get_by_name(
        introducing_file)

  if fixing_file is None:
    continue

  file_similarity_score = compute_similarity(fixing_file ,
        introducing_file)

  contribution_score = fixing_file.base_score *
        file_similarity_score
  relevance_score += contribution_score
```

b) Base score

```
summed_length = sum(fixing_commit.patches)
  for file in fixing_commit.all_files:
    if file.isJava():
      base_score = file.patch.length / summed_length
      fixing_commit.fixing_files.add(file , base_score)
```

Figure 5: Pseudo code for calculating *relevance score* (**a**) for a candidate VIC (*introducing_commit*) and calculating *base_score* (**b**) while selecting Java files. In both cases the VFC (*fixing_commit)* is given.

The similarity method (denoted as `compute_similarity` in the pseudo code) can be an arbitrary function that quantifies similarity between two texts. Here, for the sake of simplicity, we decided to use a straightforward method: we counted the ratio of identical lines in the two files after excluding empty lines.

The other operand of the product is the base score that estimates in what proportion does a file take part in the VFC (i.e. the ratio of changed lines in the file compared to the total lines changed in the fixing patch). Every corresponding file in the candidate VIC is weighted by this score as a VIC has more contribution to the vulnerability if the files it changes have greater part in the fixing (which means more changes were needed in them as part of the vulnerability fix, so they have more faulty parts). This score is determined beforehand as a result of the algorithm presented in Figure 5b. As part of calculating the base scores, we can filter on the files based on their types in the fixing commit, in this particular example, only Java files will contribute to the final relevance score of the VIC. Please note however, this part is easily changeable and as such the method can be freely extensible to any file type.

After this second, filtering phase the final output of the method is the proposed VIC dataset containing pairs of VFC and a set of VICs. Here, the set of VICs are filtered by ranking them based on relevance scores and keeping only the top *n* elements. A part of the dataset extracted this way from project-KB can be seen in Figure 6 (see Section 5 for the details).

5 RESULTS

In this section, we showcase the usage of our proposed method by presenting the tools we developed to perform the two phases: *BugIntroducerMiner* and *FilterBugIntroducer*. We also describe the dataset extracted from the project-KB database using these tools,[3] and briefly discuss the improvement of our method over simply running the plain SZZ on the fixing commits.

5.1 BugIntroducerMiner

To perform the first phase of our method (see Section 4.1), we developed a Java tool called *BugIntroducerMiner*. Information about its prerequisites and additional details (such as the exact parametrization) can be found in its `README.md` file located in the replication package.

BugIntroducerMiner is a simple Java program, which iterates over the directory structure of a project-KB like database and for each entry it runs the bug introducer finder script from the SZZ Unleashed implementation. SZZ Unleashed is basically a toolchain, using a number of Python and Java programs that, among other things, mine commits from issue trackers, filter the results or perform the bug introducing commit search. We use one of these programs, the `szz_find_bug_introducers-<version_number>.jar` file that searches for bug introducing commits.

For each invocation of the jar file, BugIntroducerMiner prepares the necessary inputs (for details, see the SZZ Unleashed repository (szz, c 14)), for example, it clones the repository containing the vulnerability and its fix. After running the program, we get the results in a JSON file called `fix_and_bug_introducing_pairs.json`. Note that this file contains the results for a single run of SZZ Unleashed but we need to run SZZ Unleashed on the whole set of VFCs and aggregate its results with BugIntroducerMiner. An example output for this process is represented in Figure 2 (the complete output might contain multiple instances of such structure).

5.2 FilterBugIntroducer

To perform the second phase of our method (see Section 4.2), we developed the tool FilterBugIntroducer, a Python program with the aim to calculate the *relevance scores* introduced in Section 4, rank the commits based on this score, and output the final VIC database. As with this other tool, information regarding the setup can be found in its `README.md` file.

The tool iterates over every CVE entry in the input VFC database and calculates the relevance scores for all the candidate VICs. To this end, for each VFC it starts iterating over the corresponding VICs. For each VIC, it calculates the similarity score for the files that are also present in the VFC. To get information about the commit and their files, the tool uses the GitHub API and some URL specific mechanisms to overcome some limitations of the API, like the limitation on commit files. The tool then aggregates the data according to the method described in Section 4.2 to calculate the overall relevance score. If the relevance score is greater than zero, the commit is considered relevant. It is important to note that for each VFC we only consider the first *m* relevant VIC, where *m* can be set by the optional parameter `--introducing-commit-limit` (default is 30). This

[3]Both the extracted VIC dataset and the tools are available publicly: https://doi.org/10.5281/zenodo.5785239

is to prevent entries with big number of VICs to run unexpectedly long.

After calculating the relevance scores, the tool selects the top *n* VICs with the highest scores, where *n* can be set by the optional parameter `--n` (default is 2). The final result will be stored in the structure shown in Figure 6 and will be saved to a YAML file named `filtered-results.yaml` by default. The example in the figure is extracted by running the tool with the top *n* parameter set to 2 (i.e. at most 2 commits with highest relevance scores are kept).

```
CVE-2008-1728:
  commitsWithIntroducers:
    c9cd1e521673ef0cccb8795b78d3cbaefb8a576a:
    - 6088e21ca06fb62790d9ea02faf8c884302e0cd9
  repo: https://github.com/igniterealtime/Openfire
CVE-2008-6505:
  commitsWithIntroducers:
    04fcefa44bae1263c7cad6986a9dafed67f0164f:
    - e05d71ba329337ba63784555fbbe9bb8e0290543
    - 78e853bcb32ea91b84a070b3d2dc03ab14bc6b23
  repo: https://github.com/apache/struts
  ...
```

Figure 6: The resulting VIC dataset structure (a YAML file).

Some notes regarding the usage of FilterBugIntroducer:

- **Caching:** The program generates heavy HTTP traffic (for the project-KB database it accesses over 130,000 HTTP URLs) mainly in the GitHub domain. To avoid unnecessary traffic and to enable multiple runs of the tool, we implemented a caching mechanism. If caching is enabled (as is by default), the tool can be re-run with different parameters without generating additional traffic (for example, to extract a new dataset with different VIC limit). However, the cache might take up considerable hard disk space (for project-KP, it takes up $\tilde{1}$3 GB).
- **Documenting:** By setting the `--document` argument to a path on the file system, the tool will output the relevance and contribution scores while calculating them for the VICs. An excerpt of such a file can be seen in Figure 3, where we see the scores generated for the VFC linked to CVE-2016-3674.

5.3 Vulnerability Introducing Commit Database from project-KB

Our final contribution is a VIC dataset extracted from the project-KB VFC database with the tools implementing our proposed method on. The VIC dataset follows the structure shown in Figure 6 containing 564 VFC entries with at most two but at least one VIC assigned to it, while the unfiletered SZZ generated dataset had VIC entries ranging from 1 to nearly 700 for each VFC. While generating the dataset, more than 110.000 files were considered (corresponding to fixing and introducing commits) from 198 open source projects.

To demonstrate our approach, we present two hand-picked examples to highlight the method's effectiveness:

- **CVE-2016-3674:** This vulnerability is already described in Section 3, however, here we elaborate further on the impact of our filtering on the SZZ extracted VIC list. Recall that the fix to this vulnerability can be seen in Figure 1. We also mentioned that SZZ Unleashed generated 17 candidate commits, two of them are shown in Figure 7. As it can be seen, in commits `deec...` and `3adb...` the changes are clearly unrelated to the vulnerability and as such their relevance scores are lower than the selected commits with high relevanace (`deec...` has a relevance score of 0.055, `3adb...` has 0.014). Moreover, in these commits only one file was changed from the 7 files that are part of the vulnerability.

  Our method choose commit `4fd3...` with the highest relevance score (see Figure 3). Figure 8 shows that it is indeed the commit that introduced the vulnerability in the `StaxDriver.java` file by instantiating an object without setting the appropriate flag. Furthermore, in this commit, two other files are also changed that contributed to the vulnerability in relevant places.
- **CVE-2016-2141:** This vulnerability is fixed in a commit (cve, c 14b) that spans through a large number of files (77) with some of them not being Java source codes (since projekt-KB is a Java vulnerability dataset, non-Java files should not be considered). Running SZZ Unleashed on this fix (as part of running BugIntroducerMiner) generates 634 candidate introducing commits. This high number of VICs is unacceptable for most of the applications, so filtering is essential. Using our method, we can conclude that the most relevant VIC has the SHA of `e2453...`[4] with a relevance score of 0.23, which indeed seems to be a good pick as it is associated with 12 vulnerable files mostly with changes that are present in the fixing commit (usually because these files are created here and the vulnerable parts have never been changed since).

[4]https://github.com/belaban/JGroups/commit/e24538a4590684d910dbdac8762c85881f519dd5

deec01beaa1bd878f7acda9f035a39238a217ae9

```
++– xstream/src/.../io/xml/StaxDriver.java


– private boolean repairingNamespace = false;

+ /**
+ * @deprecated since 1.2, use an explicit call to {
      @link #setRepairingNamespace(boolean)}
+ */
public StaxDriver(QNameMap qnameMap, boolean
      repairingNamespace) {
  – this(qnameMap, repairingNamespace, new
        XmlFriendlyReplacer());
  + this(qnameMap, new XmlFriendlyReplacer());
  + setRepairingNamespace(repairingNamespace);
  ...
```

3adb51d6c3a1a20adf88f091b200dde676d10352

```
++– xstream/src/.../io/xml/StaxDriver.java


+ import com.thoughtworks.xstream.io.
      HierarchicalStreamDriver;
import com.thoughtworks.xstream.io.
      HierarchicalStreamReader;
+ import java.io.InputStream;
+ import java.io.OutputStream;


public HierarchicalStreamReader createReader(Reader
      xml) {
  + loadLibrary();
  + try {
    +    return new StaxReader(qnameMap, createParser(
           xml));
    + }
  ...
```

Figure 7: Parts of two commits falsely identified as vulnerability introducing by SZZUnleashed for CVE-2016-3674.

4fd39f2f2616d4ea9e1d25d30dc78931be01dfb0

```
++– xstream/src/.../io/xml/StaxDriver.java


+ protected XMLInputFactory createInputFactory() {
  +    return XMLInputFactory.newInstance();
  + }
```

Figure 8: Part of an introducing commit to CVE-2016-3674 which is selected with highest relveance score.

6 THREATS TO VALIDITY

The provided vulnerability introducing dataset has not been validated manually, therefore we cannot rule out the possibility of including false positive vulnerability introducing commits. To mitigate this threat, we performed manual validation on a small random sample, which confirmed that all the included introducing commits are correct. Nonetheless, a complete manual validation is among our future plans.

The dataset contains at most two vulnerability introducing commits for each vulnerability fix. There is a chance that there are more valid introducing commits that we omit from the dataset. However, we provide the dataset extraction tools as well with which the dataset can be re-generated with adjusted number of introducing commits.

As the published vulnerability introducing dataset is extracted from the project-KB database, it's quality and accuracy influences our dataset. However, such problems in project-KB are highly unlikely as it is a manually curated dataset, therefore this threat has a very low probability.

7 CONCLUSIONS AND FUTURE WORK

In our work, we focused on source code-related vulnerability datasets, which are fundamental building blocks of vulnerability scanning and detection methods. Although datasets containing fixing patches for some vulnerabilities already exist for various programming languages, there is a lack of so-called vulnerability introducing commit datasets, which would be essential for creating and validating just-in-time vulnerability detection approaches.

To address this issue, we proposed a novel method that maps vulnerability fixing commits (VFCs) to a set of vulnerability introducing commits (VICs) using a recent implementation of the well-known SZZ algorithm. Empirical results show that applying SZZ in itself introduces a lot of false-positive commits; therefore, we extended the algorithm with an additional filtering phase. We defined a so-called relevance score for each commit that quantifies the level of connection between a fixing and an introducing commit in terms of common files and source code they affect. With this relevance score, we were able to rank introducing commits reliably and perform filtering by keeping only the highest-ranked elements.

We implemented our approach and published it as two tools (implementing the two phases) described in detail. To demonstrate the usage of these tools, we ran them on a VFC database called *project-KB* and as our main contribution, we extracted and published a new vulnerability introducing dataset based on project-KB. We manually inspected a sample of the produced results and concluded that our method: i) correctly assigns the highest scores to commits that introduce vulnerable code parts, and ii) commits ranked at the bottom are irrelevant for introducing the vulnerable behavior.

Despite the encouraging first results, there are some possible directions that we would like to address in future work:

- We choose a simple approach to measure similarity between texts. Investigating other ways for quantifying similarity between source codes could probably increase the accuracy of the method.

- Taking inspiration from the work of Tan et al. (Tan et al., 2021), the ranking could be performed with the use of machine learning models such as RankNet. It would probably increase the resource usage in exchange for a possibly more robust and accurate method.
- Manually validating all the extracted VICs would improve the confidence in the dataset quality and further strengthen the validity of our proposed method.
- Building efficient just-in-time vulnerability detection algorithms based on machine learning models trained on the extracted VICs dataset.

ACKNOWLEDGMENTS

The presented work was carried out within the SETIT project (2018-1.2.1-NKP-2018-00004),[5] supported by project TKP2021-NVA-09,[6] and the Ministry of Innovation and Technology NRDI Office within the framework of the Artificial Intelligence National Laboratory Program (MILAB). The research was partly supported by the EU-funded project AssureMOSS (Grant no. 952647) as well.

Furthermore, Péter Hegedűs was supported by the Bolyai János Scholarship of the Hungarian Academy of Sciences and the ÚNKP-21-5-SZTE-570 New National Excellence Program of the Ministry for Innovation and Technology.

REFERENCES

(2021. dec. 14.a). Cve 2016-3674: https://nvd.nist.gov/vuln/detail/cve-2016-3674.

(2021. dec. 14.b). Jgroups fixing commit - https://github.com/belaban/jgroups/commit/38a882331035ffed205d15a5c92b471fd09659c.

(2021. dec. 14.). Sap - project kb: https://github.com/sap/project-kb/tree/master/vulnerability-data.

(2021. dec. 14.). The state of open source vulnerabilities 2021: https://www.whitesourcesoftware.com/resources/research-reports/the-state-of-open-source-vulnerabilities/.

[5]Project no. 2018-1.2.1-NKP-2018-00004 has been implemented with the support provided from the National Research, Development and Innovation Fund of Hungary, financed under the 2018-1.2.1-NKP funding scheme.

[6]Project TKP2021-NVA-09 was implemented with the support provided by the Ministry of Innovation and Technology of Hungary from the National Research, Development and Innovation Fund, financed under the TKP2021-NVA funding scheme.

(2021. dec. 14.). Szz unleashed:https://github.com/ wogscpar/szzunleashed.

(2021. dec. 14.). Xstream: https://github.com/x-stream/xstream.

(2021. nov. 20.). The mitre corporation - common vulnerabilities and exposures: https://www.cve.org/.

(2021. nov. 20.). U.s. national institute of standards and technology - national vulnerability database: https://nvd.nist.gov/.

Amin, A., Eldessouki, A., Magdy, M. T., Abdeen, N., Hindy, H., and Hegazy, I. (2019). Androshield: Automated android applications vulnerability detection, a hybrid static and dynamic analysis approach. *Information*, 10(10).

Bhandari, G. P., Naseer, A., and Moonen, L. (2021). Cvefixes: Automated collection of vulnerabilities and their fixes from open-source software. *CoRR*, abs/2107.08760.

Borg, M., Svensson, O., Berg, K., and Hansson, D. (2019). Szz unleashed: an open implementation of the szz algorithm - featuring example usage in a study of just-in-time bug prediction for the jenkins project. *Proceedings of the 3rd ACM SIGSOFT International Workshop on Machine Learning Techniques for Software Quality Evaluation - MaLTeSQuE 2019*.

Cao, S., Sun, X., Bo, L., Wei, Y., and Li, B. (2021). Bgnn4vd: Constructing bidirectional graph neural-network for vulnerability detection. *Information and Software Technology*, 136:106576.

Dai, J., Zhang, Y., Jiang, Z., Zhou, Y., Chen, J., Xing, X., Zhang, X., Tan, X., Yang, M., and Yang, Z. (2020). *BScout: Direct Whole Patch Presence Test for Java Executables*. USENIX Association, USA.

Falleri, J.-R., Morandat, F., Blanc, X., Martinez, M., and Monperrus, M. (2014). Fine-grained and accurate source code differencing. *ASE 2014 - Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*.

Gkortzis, A., Mitropoulos, D., and Spinellis, D. (2018). Vulinoss: A dataset of security vulnerabilities in open-source systems. In *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*, pages 18–21.

Herzog, S. (2010). Xml external entity attacks (xxe). *Retrieved October*, 13:2013.

Li, F. and Paxson, V. (2017). A large-scale empirical study of security patches. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, page 2201–2215, New York, NY, USA. Association for Computing Machinery.

Li, H., Kim, T., Bat-Erdene, M., and Lee, H. (2013). Software vulnerability detection using backward trace analysis and symbolic execution. In *2013 International Conference on Availability, Reliability and Security*, pages 446–454.

Meneely, A., Srinivasan, H., Musa, A., Tejeda, A. R., Mokary, M., and Spates, B. (2013). When a patch goes bad: Exploring the properties of vulnerability-contributing commits. In *2013 ACM / IEEE Interna-*

tional Symposium on Empirical Software Engineering and Measurement, pages 65–74.

Meneely, A., Tejeda, A. C. R., Spates, B., Trudeau, S., Neuberger, D., Whitlock, K., Ketant, C., and Davis, K. (2014). An empirical investigation of socio-technical code review metrics and security vulnerabilities. In *Proceedings of the 6th International Workshop on Social Software Engineering*, SSE 2014, page 37–44, New York, NY, USA. Association for Computing Machinery.

Meneely, A. and Williams, O. (2012). Interactive churn metrics: socio-technical variants of code churn. *ACM SIGSOFT Software Engineering Notes*, 37:1–6.

MITRE Corporation (2021. nov. 21.). CVE - Common Vulnerabilities and Exposures. https://cve.mitre.org/. [Online; accessed 29-April-2020].

Perl, H., Dechand, S., Smith, M., Arp, D., Yamaguchi, F., Rieck, K., Fahl, S., and Acar, Y. (2015). Vccfinder: Finding potential vulnerabilities in open-source projects to assist code audits. In Ray, I., Li, N., and Kruegel, C., editors, *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-16, 2015*, pages 426–437. ACM.

Ponta, S. E., Plate, H., Sabetta, A., Bezzi, M., and Dangremont, C. (2019). A manually-curated dataset of fixes to vulnerabilities of open-source software. In *Proceedings of the 16th International Conference on Mining Software Repositories*.

Sliwerski, J., Zimmermann, T., and Zeller, A. (2005). When do changes induce fixes? volume 30.

Tan, X., Zhang, Y., Mi, C., Cao, J., Sun, K., Lin, Y., and Yang, M. (2021). Locating the security patches for disclosed oss vulnerabilities with vulnerability-commit correlation ranking. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, CCS '21, page 3282–3299, New York, NY, USA. Association for Computing Machinery.

Wang, X., Wang, S., Feng, P., Sun, K., and Jajodia, S. (2021a). Patchdb: A large-scale security patch dataset. In *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 149–160.

Wang, X., Wang, S., Feng, P., Sun, K., Jajodia, S., Benchaaboun, S., and Geck, F. (2021b). Patchrnn: A deep learning-based system for security patch identification.

Woo, S., Park, S., Kim, S., Lee, H., and Oh, H. (2021). Centris: A precise and scalable approach for identifying modified open-source software reuse. In *Proceedings of the 43rd International Conference on Software Engineering*, ICSE '21, page 860–872. IEEE Press.

Xiao, Y., Chen, B., Yu, C., Xu, Z., Yuan, Z., Li, F., Liu, B., Liu, Y., Huo, W., Zou, W., and Shi, W. (2020). MVP: detecting vulnerabilities using patch-enhanced vulnerability signatures. In Capkun, S. and Roesner, F., editors, *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, pages 1165–1182. USENIX Association.

Zheng, Y., Pujar, S., Lewis, B. L., Buratti, L., Epstein, E. A., Yang, B., Laredo, J., Morari, A., and Su, Z. (2021). D2A: A dataset built for ai-based vulnerability detection methods using differential analysis. *CoRR*, abs/2102.07995.