PatchDB: A Large-Scale Security Patch Dataset

Xinda Wang†, Shu Wang†, Pengbin Feng*, Kun Sun, Sushil Jajodia
Center for Secure Information Systems, George Mason University, Fairfax, VA, USA
{xwang44, swang47, pfeng4, ksun3, jajodia}@gmu.edu

***Abstract*—Security patches, embedding both vulnerable code and the corresponding fixes, are of great significance to vulnerability detection and software maintenance. However, the existing patch datasets suffer from insufficient samples and low varieties. In this paper, we construct a large-scale patch dataset called PatchDB that consists of three components, namely, NVD-based dataset, wild-based dataset, and synthetic dataset. The NVD-based dataset is extracted from the patch hyperlinks indexed by the NVD. The wild-based dataset includes security patches that we collect from the commits on GitHub. To improve the efficiency of data collection and reduce the effort on manual verification, we develop a new nearest link search method to help find the most promising security patch candidates. Moreover, we provide a synthetic dataset that uses a new oversampling method to synthesize patches at the source code level by enriching the control flow variants of original patches. We conduct a set of studies to investigate the effectiveness of the proposed algorithms and evaluate the properties of the collected dataset. The experimental results show that PatchDB can help improve the performance of security patch identification.**

***Index Terms*—security patch, open source software, dataset**

I. Introduction

A security patch is a set of changes on source code to fix the vulnerability. Both vulnerable code and the corresponding fixes are embedded in security patches. Compared to non-security patches (e.g., performance bug fixes and new features), security-related patches usually take higher precedence to be applied. Hence, security patch identification plays a significant role in security research, especially in vulnerability mitigation and software maintenance. The verified security patches can be used to generate signatures for detecting more vulnerabilities or patch presence [17], [36], [40].

A straightforward method to identify security patches is to analyze the literal descriptions (e.g., bug reports and commit messages) using text-mining techniques [15], [16], [26], [43]. However, such identification methods are error-prone due to the poor quality of the textual information. For instance, 61% of security patches for the Linux kernel do not mention security impacts in their description or subjects [35]. Instead, other techniques go a further step by analyzing the source code of security patches [29], [39], [42]. Nevertheless, this process requires considerable human effort and expertise. Although some automatic security patch identification tools have been proposed [31]–[33], they suffer from performance and generalization issues.

There is an increasing demand for the deployment of a robust classifier (e.g., deep learning model). To achieve this goal, one of the biggest challenges is the lack of sufficient patch samples in the model learning and testing stages. Most of the existing patch datasets [18], [20]–[22], [36] have several limitations. First, the number of publicly available security patches is not large enough to train the model. Second, those security patches are collected from single or several specific software repositories, leading to biases towards certain types of software and vulnerabilities. Finally, they focus on specific types of security patches (e.g., patches of sanity testing), which limits the generalization capability of the learned models. As a result, all these existing public datasets fail to involve complex and variant patches for learning a general classifier. Also, the empirical study over those patch datasets may be biased.

In this paper, we construct a large-scale patch dataset called PatchDB that contains a large number of security patches and non-security patches in C/C++ languages. It consists of three datasets, namely, *NVD-based dataset*, *wild-based dataset*, and *synthetic dataset*. PatchDB not only contains the verified security patches indexed by the National Vulnerability Database (NVD) [5] but also includes a large number of patches obtained from the wild. Moreover, to enrich the patch variants, PatchDB also provides an additional synthetic patch dataset that is automatically generated from the existing samples via a new patch oversampling technique. In contrast, we call the patches in both the NVD-based dataset and the wild-based dataset as natural patches.

We first construct the NVD-based dataset based on the NVD, the largest known source for extracting security patches [20]. In this dataset, around 4K security patches are collected by crawling reference hyperlinks provided by the NVD. Note that we focus on the patches in C/C++ projects that have the largest number of vulnerabilities. The NVD-based dataset contains many samples of typical severe vulnerabilities, which are useful for security patch studies.

The patches in the wild-based dataset are collected from the commits on GitHub. It is well known that around 6-10% of commits are security patches that are not reported to the NVD [20], [32]; however, it is time consuming and labor intensive to manually check if each commit is security-related. Based on the assumption that in the feature space, the closer a commit sample is to a verified security patch, the more likely it is a security one, we develop a *nearest link search* algorithm to find an equal number of candidates from the GitHub commits that are closest to the security patches in the NVD-based dataset. Then, each candidate is manually verified as either a security patch or non-security patch by three security experts who cross-check their decisions. After five rounds of this

†The first two authors contributed equally to this work.
*Corresponding author: Pengbin Feng

data augmentation process, we finally collect 8K new security patches and 23K cleaned non-security patches in the wild-based dataset. Our experiments show that the proportion of security patches in the candidates identified by the nearest link search algorithm is around 30%, which is three times better than the brute force search (i.e., 6-10%).

PatchDB also provides a synthetic patch dataset, which is generated based on the above two natural datasets. It is inspired by the fact that some vulnerability detection studies use artificial vulnerabilities (e.g., SARD [8]) to train their deep learning-based models due to the limited vulnerable code gadgets [22], [23]. Similarly, patch synthesis could be a useful approach to help improve the security patch identification or analysis. However, there is no synthetic patch dataset or patch synthesis algorithm publicly available. Therefore, we further develop an oversampling method to synthesize patches. Different from traditional oversampling techniques [11] that only synthesize instances in the feature space, our method generates a set of synthetic patches by modifying the critical statements at the source code level. Since around 70% security patches involve modifications of conditional statements (i.e., `if` statements) [24], we focus on enriching the control flow variants of original patches. Specifically, we design eight variants for `if` statements without affecting the original program functionality. We develop a tool to automatically synthesize patches based on these variants. The experimental results show that synthesizing patches for a limited-size dataset could improve the performance of automatic security patch identification.

Overall, our PatchDB dataset has the following distinctive features: 1) it is a **large-scale** security patch dataset that contains 12K natural security patches, where 4K are from the NVD-based dataset and 8K are from the wild-based dataset; 2) it covers **various** types of security patches in terms of code changes; 3) it provides a **cleaned** non-security patch dataset of 23K instances; 4) it provides a **synthetic** dataset where the patches are synthesized from the NVD-based dataset and wild-based dataset; and 5) each natural patch is **accessible** on GitHub for further context information. As far as we know, PatchDB is the largest dataset that contains NVD-based, wild-based, and synthetic security patches. Also, we find the 8K security patches in the wild-based dataset are silently published, i.e., not listed in any CVE entries.

Moreover, we conduct a set of experimental studies on the composition and quality of PatchDB as well as the effectiveness of our proposed algorithms. We first show that nearest link search can help dramatically reduce human efforts on identifying security patches from the wild. Also, a larger search range (i.e., more unlabeled GitHub commits) can increase the identification efficiency. Second, our dataset augmentation method outperforms state-of-the-art machine learning techniques by providing better tolerance to the distribution discrepancy between the NVD and the wild patches. Third, our experimental results show that synthetic patches can effectively increase the complexity and variance of a limited-size dataset. Moreover, we further study the dataset composition by classifying security patches into multiple categories in terms of code changes. The categorization results of the NVD-based dataset exhibit a long tail distribution with the high imbalance and our dataset augmentation approach can alleviate the imbalance by introducing more instances in the tail. Finally, we verify the usefulness of PatchDB by showing that the performance of automatic patch analysis can be improved by adopting the large-scale PatchDB.

In summary, we make the following contributions:

- We construct a large-scale patch dataset called PatchDB that includes the NVD-based dataset, wild-based dataset, and synthetic dataset. To the best of our knowledge, we are the first to collect and release a diverse set of patches at this scale[1].
- We develop a dataset augmentation scheme by finding the most promising security patch candidates using a new nearest link search algorithm, which can achieve better performance than the state-of-the-art approaches by reducing around 66% efforts on human verification.
- We propose a new oversampling technique to synthesize patches at the source code level. The experimental results show that synthetic patches are effective for automatic patch analysis tasks.
- We conduct an empirical study on PatchDB by categorizing security patches based on their code changes. We also assess the dataset quality and obtain some interesting observations.

II. Background

In this section, we give the definition of software patches and illustrate the differences between security and non-security patches. We also introduce the NVD, which is a reliable repository for us to extract security patches.

A. Security and Non-Security Patches

A software patch is a set of changes between two versions of source code to improve security, resolve functionality issues, and add new features. Security patches address specific security vulnerabilities, enhancing the security of the software. Non-security patches include bug fix patches and new feature patches. The bug fix patches make the software run more smoothly and reduce the likelihood of a crash by correcting the software bugs. The new feature patches add new or update existing functionality to the software.

On the version control platform like GitHub [3], a commit can be regarded as a patch. Listing 1 and 2 show an example of security patch and non-security patch downloaded from GitHub, respectively. Each patch is identified by a 20-byte long hash string and the modified file will be recognized by a line starting with *diff --git*. The consecutive removed and added statements (i.e., lines start with − or +) in one patch are called one hunk. Around the hunk, there are typically several context lines. One patch may contain more than one hunk over multiple functions and files. Listing 1 is a security patch for

[1]The dataset is available at https://github.com/SunLab-GMU/PatchDB.

```
1  commit b84c2cab55948a5ee70860779b2640913e3ee1ed
2  diff --git a/src/bits.c b/src/bits.c
3  index 014b04fe4..a3692bdc6 100644
4  --- a/src/bits.c
5  +++ b/src/bits.c
6  @@ -953,7 +953,7 @@ bit_write_UMC (Bit_Chain *dat,
       BITCODE_UMC val)
7        if (byte[i] & 0x7f)
8          break;
9
10 -    if (byte[i] & 0x40)
11 +    if (byte[i] & 0x40 && i > 0)
12       i--;
13     byte[i] &= 0x7f;
14     for (j = 4; j >= i; j--)
15 }
```

Listing 1: An example of security patch for a stack underflow vulnerability (CVE-2019-20912).

```
1  commit c3b3c274cf7911121f84746cd80a152455f7ec97
2  diff --git a/main.c b/main.c
3  index 6a3eee2eb..b8ad59018 100644
4  --- a/main.c
5  +++ b/main.c
6  @@ -575,5 +575,8 @@ finish:
7
8            dbus_shutdown();
9
10 +          if (getpid() == 1)
11 +                  freeze();
12 +
13           return retval;
14   }
15
16 }
```

Listing 2: An example of non-security patch in *systemd*.

vulnerability CVE-2019-2091 that prevents stack underflow in the function `bit_write_UMC` (as identified in Line 6) by replacing the previous incomplete check (Line 10) with one more sanity check for the local variable `i` (Line 11), which is used as an index to access the array `byte` (Line 13). By adding a check of *pid*, the non-security patch in Listing 2 only freezes the *init* process (Line 10-12) but exits all other processes, avoiding a potential crash.

B. NVD

The National Vulnerability Database (NVD) [5] is the largest publicly available source of vulnerability intelligence maintained by the U.S. National Institute of Standards and Technology (NIST). Besides synchronizing with the Common Vulnerabilities and Exposures (CVE) system [2] where a CVE-ID is assigned to each vulnerability, the NVD provides enhanced information such as patch availability, severity scores, and impact ratings. For each CVE entry, the NVD provides external reference URLs of advisories, solutions, tools, etc. Among them, security patches for the current vulnerability could be extracted from a hyperlink tagged with "patch" (if any). By crawling such hyperlinks, it is possible to extract security patches and even access the corresponding source code repositories. However, due to the limited human power supporting the NVD, the patch information may not be available or accurate, and some CVE entries in the NVD are not provided with any links for the patch.

Since not all known vulnerabilities are reported to the CVE or accepted by the CVE Numbering Authorities (CNAs) [22], [32], [37], the security patches of those vulnerabilities cannot be retrieved from the NVD. On the other side, it also means that there exist a number of security patches in the wild that can be used to enlarge the security patch dataset.

III. Methodology

Figure 1 shows the methodology of constructing the PatchDB, which consists of three components, namely, *NVD-based patch dataset*, *wild-based patch dataset*, and *synthetic patch dataset*. First, we build an initial security patch dataset by crawling the NVD entries that have corresponding patch hyperlinks. Since the number of NVD security patches (around 4K) is not large enough to train a robust classifier (e.g., deep learning models), we develop a novel augmentation approach named *nearest link search* to help increase the efficiency of discovering security patches from the wild (i.e., GitHub). Moreover, we propose a new oversampling method to synthesize security patches at the source code level.

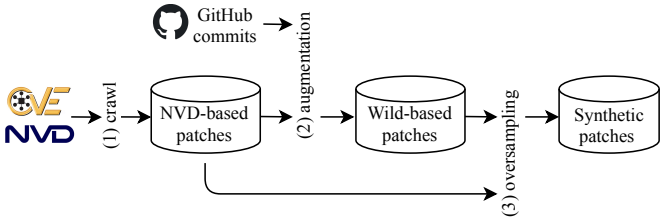

Fig. 1: PatchDB construction methodology.

A. Extracting Security Patches from the NVD

The first step of constructing PatchDB is to collect and screen the security patches that have already been indexed by the NVD. As the largest publicly available source of vulnerability intelligence, the NVD provides pertinent hyperlinks of corresponding patches for a portion of the CVE entries. We focus on the software repositories hosted on GitHub, where each patch (commit) is identified with a unique hash value. We observe that the URLs of these patches are usually in this form: *https://github.com/{owner}/{repo}/commit/{hash}*. By downloading all these links with a suffix *.patch*, we can obtain thousands of security patches associated with CVE IDs. We focus on patches of projects written in C/C++ that are the languages with the highest number of vulnerabilities [34].

However, the patches of C/C++ projects may contain modifications on files such as *.changelog*, *.kconfig*, *.sh*, *.phpt*, etc. After manually checking a random subset of them, we find most of these non-C/C++ files are documentations or changes corresponding to modifications of C/C++ files (*.c*, *.cpp*, *.h*, and *.hpp*) and they do not play an important role in fixing vulnerabilities. Therefore, we need to remove these non-C/C++ parts from patches. In this way, we obtain a dataset of 4076 security patches from 313 GitHub repositories between 1999 and 2019, which is by far the largest security patches in C/C++ collected from the NVD[2].

[2]The 4K security patches collected by other work [20] contain multiple types of programming languages and the dataset is not publicly available.

We also need to collect a dataset of non-security patches, which is useful to develop and evaluate a machine learning model to identify security patches. We first download these 313 GitHub repositories whose patch information is available in the NVD. Then we acquire all the commits with the command *git log*. However, we cannot assume all these commits except the above 4076 security patches would be cleaned non-security patches. Actually, after reviewing some random subsets of those commits, we observe that around 6-10% of them are security patches, which is consistent with some previous studies [20], [22], [32]. Inspired by the existence of those silent security patches (i.e., patches not reported to CVE) in the wild, we consider commits on GitHub as a good source to enlarge the security patches in the NVD-based dataset.

B. Augmenting Patch Dataset via Nearest Link Search

Though we find 6-10% of patches in the wild are security patches, it is still labor-intensive and lacks the efficiency to manually screen out these security patches. To reduce the search range of potential candidates, we develop a dataset augmentation method to help find the most likely security patch candidates from the wild patches. The benefit of our method is twofold. First, we can enlarge the security patch dataset. Second, we can clean the hidden security patches from the non-security patch dataset.

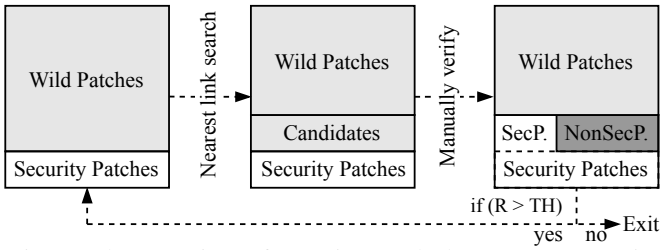

Fig. 2: The overview of security patch dataset augmentation (the candidates are selected from wild patches and would be verified manually by the professional security researchers).

Overview of Dataset Augmentation. Figure 2 shows our dataset augmentation scheme, which consists of three steps: *candidate selection*, *manual verification*, and *loop judgment*.

First, we propose a nearest link search algorithm to better select security patch candidates from the wild using the features derived from already labeled security patches in our NVD-based dataset. The algorithm selects the candidate set that has the global minimal distance with the set of verified security patches in the feature space. It is based on one observation that the closer a sample is to a verified security patch sample, the more likely it is a security patch.

Second, these candidates will be manually verified by professional security researchers. Since the nearest link search algorithm can narrow down the promising candidates to be verified, the labor costs would be reduced dramatically.

Finally, we evaluate the proportion R of security patches in the candidates and repeat the above procedures with the enlarged security patch dataset if R is larger than a threshold, where the proportion R implies ample security patches remaining in the wild set. When R drops below a threshold, we exit the dataset augmentation process.

TABLE I: List of features for nearest link search.

No.	Feature Descriptions
1	# changed lines
2	# hunks
3-6	# added/removed/total/net lines
7-10	# added/removed/total/net characters
11-14	# added/removed/total/net if statements
15-18	# added/removed/total/net loops
19-22	# added/removed/total/net function calls
23-26	# added/removed/total/net arithmetic operators
27-30	# added/removed/total/net relation operators
31-34	# added/removed/total/net logical operators
35-38	# added/removed/total/net bitwise operators
39-42	# added/removed/total/net memory operators
43-46	# added/removed/total/net variables
47-48	# total/net modified functions
49-51	mean/min/max Levenshtein distance within hunks†
52-54	mean/min/max Levenshtein distance within hunks⋆
55	# same hunks†
56	# same hunks⋆
57-58	# and % of affected files
59-60	# and % of affected functions

† Before token abstraction. ⋆ After token abstraction.

Algorithm of Nearest Link Search. Now we detail the nearest link search algorithm, which is the core of our method. Given a set of security patches, the algorithm can find an equal number of candidates from the wild dataset in three steps: *feature space construction*, *weighted distance matrix calculation*, and *nearest link optimization*.

1) Feature Space Construction: The feature space is constructed based on the syntactic features extracted from the source code of patches. These features, e.g., the conditional statement amount and loop statement amount, can reveal the differences between security patches and non-security patches. Our hypothesis is that the patches with similar syntactic features tend to have similar properties and semantics.

Table I lists 60 types of features used in our feature space construction. These features can be divided into three types. Features 1-10 are basic patch features that indicate the text-level changes. Features 11-56 indicate the changes in the programming language level that are language-dependent. Features 57-60 reveal the affected range by the patch. Since the patch is not a complete program unit and contains both pre-patched and post-patched code, we implement a parser to extract these features using Python. For both the security patches and wild patches, the 60-dimensional features are extracted to construct the feature space for further processing.

2) Weighted Distance Matrix: The similarity of the two patches is represented as the distance of corresponding features in the feature space. However, since the extracted features have different scales in different dimensions, it is necessary to normalize each dimension in features with an appropriate weight. For the j-th feature in the i-th patch, we normalize the feature a_{ij} as

$$a'_{ij} = a_{ij} \cdot w_j = a_{ij} \cdot \frac{1}{max|\boldsymbol{a}_j|},$$

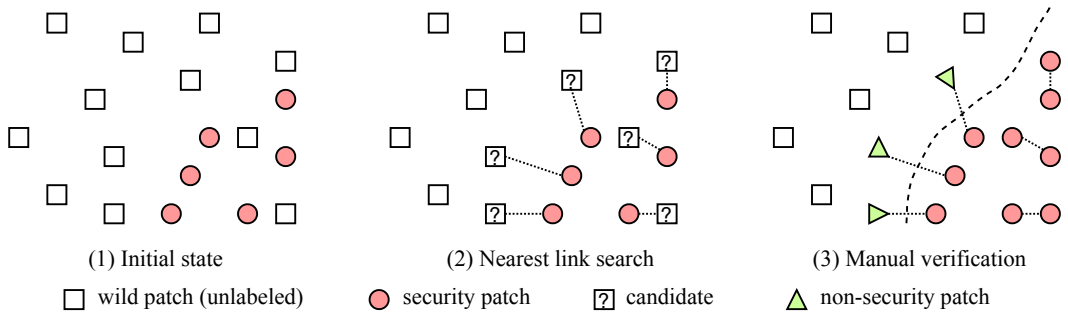

Fig. 3: The nearest link search and candidate verification.

where w_j is the weight for the j-th feature, $\boldsymbol{a}_j$ is the vector that refers to the j-th features of all patches. The normalized features would be in the range of -1 to 1 after the weighting so that the distances in different dimensions can be comparable and the information of net values for some features (e.g., net modified functions) will be reserved.

Then, we can calculate the Euclidean distance d_{mn} between the m-th security patch and the n-th wild patch, so as to build a weighted distance matrix $\boldsymbol{D} = \{d_{mn}\}^{M \times N}$, where M and N refer to the security patch number and wild patch number.

3) Nearest Link Optimization: The goal of the nearest link search is to find a wild patch for each verified security patch so that the total distance of all pairs would be a minimum. A pair of one security and one non-security patch is also called a link, which presents the selected wild patch has a high similarity with the verified security patch. Thus, the set of the selected wild patches becomes our candidate set. Because one wild patch can only be linked to up to one security patch, the size of the candidate patches is the same as the size of the verified security patches.

As shown in Figure 3, the candidate patches are a set of patches located by the nearest link and they have similar features with verified security patches. Therefore, these candidates have a higher probability of being verified as security patches compared with other wild patches. After human verification, some candidates would be labeled as a security patch and added to the existing security patch dataset. Other candidates would be added to the non-security patch dataset. In other words, the size of the verified security patches will keep increasing when new security patches are identified from the wild. If a candidate is verified as a non-security patch, this instance could also increase the discernible capability of the patch classifier near the decision boundary. That is because this patch link crosses the decision boundary and reveals the data distribution around the region as well.

To compute the nearest link, we convert the candidate search problem into an optimization problem. The optimization objective is to minimize the sum of the distances in each link, as shown in the following formula:

$$min \sum_{m=1}^{M} d_{mc_m}, \quad s.t.\ c_m \in \mathbb{Z} \cap [1, N],\ c_1 \neq c_2 \neq .. \neq c_M.$$

where d_{mc_m} refers to the distance of the m-th patch link, and c_m is the index of the wild patch that is linked to the m-th verified security patch. The optimization problem is to obtain the set $\{c_m\}_{m=1}^{M}$ with M different elements to minimize the total link distance. It is similar to the Kuhn–Munkres (KM) algorithm [12] that seeks the combinatorial optimization in the assignment problem, hence it is hard to find the globally optimal solutions. To solve this problem, we adopt an approximately optimal solution with a greedy algorithm. The algorithm is illustrated in Algorithm 1, where the time complexity is $O(MN^2)$. Note that our nearest link search is different from the k-nearest neighbors (KNN) algorithm where K candidates will be selected according to one verified sample and one candidate may be assigned to multiple verified samples even if $K = 1$. In the nearest link, each candidate can only be selected at most once, and each selected candidate will be paired with one individual verified sample.

C. Generating Synthetic Dataset via Oversampling

It is known that when developing learning based patch analysis models, the training phase may face two challenges, namely, the model over-fitting problem due to insufficient

Algorithm 1 The Nearest Link Search Algorithm

Input: the weighted distance matrix $\boldsymbol{D} = \{d_{mn}\}^{M \times N}$
Output: the index set for selected wild patches $\{c_m\}_{m=1}^{M}$

```
/* init the minimum index */
U = {u_1, u_2, ..., u_M},  u_m = min{d_mn}_{n=1}^N
V = {v_1, v_2, ..., v_M},  v_m = argmin_n{d_mn}_{n=1}^N
/* find the index */
C = {c_1 = 0, c_2 = 0, ..., c_M = 0}
for i ← 1 to M do
    m_0 ← argmin U
    n_0 ← v_{m_0}
    /* if n_0 has been used */
    if n_0 ∈ C then
        l = {d_{m_0 n}}_{n=1}^N
        for j ← 1 to M do
            if c_j ≠ 0 then
                l_{c_j} ← inf
        n_0 ← argmin l
    c_{m_0} ← n_0
    u_{m_0} ← inf
output C
```

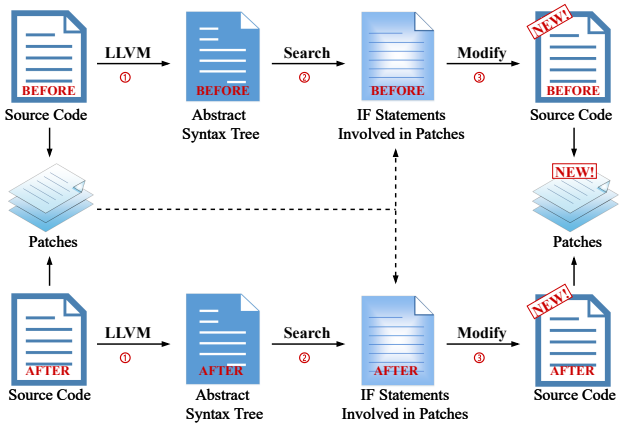

Fig. 4: The overview of oversampling at source code level.

training instances and the misleading fortuitous patterns. To alleviate these problems, some artificial code gadget datasets (e.g., SARD [8]) are adopted during the training [22], [23]. However, since patches are not complete program units and contain both pre-patched and post-patched code at the same time, previous code synthesis algorithm [10] cannot be applied to create artificial patches. Therefore, we propose a new oversampling algorithm to generate artificial patches at the source code level, which is different from traditional oversampling [11] that synthesizes instances in the feature space. We provide more interpretability since vector instances generated by traditional methods cannot be transformed back to the source code patch. As around 70% security patches involve modifications that add or update conditional statements (i.e., `if` statements) [24], we focus on enriching the control flow variants of natural patches. Note that we refer to real patches as natural patches to distinguish from synthetic ones.

Figure 4 depicts the overview of our oversampling method, which contains three steps. First, for a given patch, we generate the Abstract Syntax Trees (ASTs) from its related source code files. Second, among these ASTs, we locate the `if` statements involved with code changes in the patch. Third, given existing patches, we transform their `if` statements according to a set of predefined variant templates in order to get the corresponding artificial patches. We detail each step in the following.

1) Generating ASTs from Patches: Since the patch is a bunch of differences between two versions of files, it is not a complete top-level program unit and some related portions may be missing. Therefore, we cannot directly generate ASTs from patches. Instead, for each patch, we retrieve the related files before and after applying the target patch so that these corresponding files can be parsed. Since we have downloaded all the repositories associated with our patch dataset and each patch can be uniquely identified with its commit hash value, we can easily roll back the corresponding repository to the point just before and after committing the target patch. Furthermore, we can easily find out the patch-related files that are listed in lines that start with *diff --git*. Then, we use LLVM [4] to generate the ASTs for these files.

2) Locating Conditional Statements: The goal of this step is to locate all the `if` statements related to the patch, i.e., `if` statements that are added, deleted, or modified by the patch. The `if` statements can be located by utilizing the `IfStmt <line:N:N, line:N:N>` field in the AST files. From the ASTs, we can retrieve the key information of `if` statements, such as the start line, end line, and the internal structure. In the next step, our transformation would focus on these `if` statements since they are more likely to embed critical changes of security patches.

3) Adding Control Flow Variants: Instead of modifying both the BEFORE version and AFTER version at the same time, we can modify one of these two versions and generate patch variants. When we modify the AFTER version source code, it is equivalent to adding some additional modifications to the AFTER version code. In that case, when considering the patch variant, it is equivalent to the patch between the original BEFORE version and the new AFTER version (i.e., original AFTER version plus additional modifications). Therefore, we only need to merge the original patch and the additional modifications. Similarly, when we only change the BEFORE version source code, it is equivalent to adding some additional modifications to the original BEFORE version code. Therefore, the patch variant is equivalent to the merge of the *inverse* additional modifications and the original patch.

```
const int _SYS_ZERO = 0;
if (_SYS_ZERO || IF_STMT) {}
```

```
const int _SYS_ONE = 1;
if (_SYS_ONE && IF_STMT) {}
```

```
bool _SYS_STMT = IF_STMT;
if (True == _SYS_STMT) {}
```

```
bool _SYS_STMT = !(IF_STMT);
if (!_SYS_STMT) {}
```

```
int _SYS_VAL = 0;
if (IF_STMT){_SYS_VAL = 1;}
if (_SYS_VAL) {}
```

```
int _SYS_VAL = 1;
if (IF_STMT){_SYS_VAL = 0;}
if (!_SYS_VAL) {}
```

```
int _SYS_VAL = 0;
if (IF_STMT){_SYS_VAL = 1;}
if (_SYS_VAL && IF_STMT) {}
```

```
int _SYS_VAL = 1;
if (IF_STMT){_SYS_VAL = 0;}
if (!_SYS_VAL || IF_STMT) {}
```

Fig. 5: Eight different variants of IF statements.

As shown in Figure 5, we apply eight types of variants on `if` statements to generate the synthetic patches. The statements in normal font are control flow related contents in natural patches, and the statements in bold are flow variants added to generate artificial patches. By introducing control flow complexity to natural patches, the synthetic dataset can enrich representations of patches and alleviate the over-fitting of the learning model, improving the performance of automatic patch analysis tasks.

IV. Evaluation

Based on the methodology introduced in Section III, we collect a large-scale security patch dataset consisting of NVD-based patches, wild-based patches, and synthetic patches. We conduct a set of experimental studies to investigate the effectiveness of the proposed algorithms and evaluate the properties of the collected dataset. Specifically, our evaluation aims to answer the following five research questions (RQs).

- **RQ1**: How to efficiently construct the wild-based security patch dataset using the nearest link search approach?
- **RQ2**: What is the performance of the nearest link search approach compared with the state-of-the-art work?
- **RQ3**: How useful are the synthetic security patches?
- **RQ4**: What is the composition of our collected dataset?
- **RQ5**: What is the quality of PatchDB?

A. Wild-based Dataset Construction (RQ1)

Based on the NVD-based dataset that includes 4076 security patches, we construct the wild-based security patch dataset by identifying security patches from the wild using the proposed nearest link search and human-in-the-loop approaches. Since it is difficult to download all commits of every GitHub repository, we focus on 313 GitHub repositories that are included in the NVD reference hyperlinks and consider all their commits as the wild. In total, we collect 6M patches in the wild.

Table II presents the dataset augmentation setup and results in five rounds. The first two columns show the size of unlabeled wild data used to perform the nearest link search and the corresponding round number, respectively. The third column is the number of candidates identified in each round. Since our nearest link search method locates its nearest neighbor for each known security patch, this number is equal to the number of already labeled security patches. The fourth column exhibits the number of real security patches verified by security experts. To ensure the correctness of manual verification, three authors of this paper label the candidates separately and then cross-check their labeling results. The last column shows the ratio of the verified security patches (column 4) to the candidates (column 3). The higher ratio means the higher efficiency.

In the beginning, the NVD-based dataset includes 4076 security patches, and we use it as the initial dataset to search their nearest links in an unlabeled wild dataset. Since it is too expensive to compute distances from all the 6M instances in the wild, we construct a smaller Set I by randomly selecting 100K commits from the wild 6M instances. We run three rounds of dataset augmentation over Set I. In the first round, the nearest link search method generates 4076 candidates, and we manually verify that 895 are security patches, a ratio of 22%. Now we have 4971 security patches (i.e., 4076 NVD-based and 895 wild-based ones). Meanwhile, 3181 candidates are labeled as non-security patches through the manual verification process, and we remove all these labeled patches from Set I. In the second round, based on 4971 labeled security patches, we repeat the above procedures on the updated Set I, and 1235 instances out of 4971 candidates are manually verified as security patches (a ratio of 25%). Similarly, in the third round, we manually identify 993 new security patches from 6206 candidates, and the ratio drops to 16%. Since there can be 6-10K security patches in the 100K search range and only around 1K instances are identified in each round, a large number of unexplored patches still remain. In such cases, ratios may not definitely decrease after each round.

While the ratios in Set I are 16-25%, we wonder if the ratio can be further increased. Intuitively, the ratio may increase if the candidates are located in a larger unlabeled wild dataset since it is more likely to contain security patches that are more similar to existing security patch instances. Therefore, instead of continuing in Set I, we conduct the security patch dataset augmentation in a larger unlabeled wild dataset Set II, another 200K randomly selected from the 6M GitHub commits. Among 7199 candidates selected by the nearest link search, 2088 instances are security patches and the total number of known security patches increases to 9287 after Round 4. We find that the ratio (29%) is higher than the first three rounds, which means a larger search range can enable a higher ratio. To verify this, we conduct another round (Round 5) of data augmentation on Set III with another 200K randomly selected instances. We discover 2786 security patches from 9287 candidates (30%). It confirms that the ratio increases along with a larger search range. Compared with the brute force search that considers all the unlabeled data as candidates where only 6-10% candidates are security patches, our method can almost triple the efficiency of human verification, in other words, reduce around 66% efforts.

TABLE II: # of security patches identified in five rounds.

Search Range (unlabeled patches)	Round	Candidates	Verified Security Patches	Ratio
Set I: 100K	1	4076	895	22%
	2	4971	1235	25%
	3	6206	993	16%
Set II: 200K	4	7199	2088	29%
Set III: 200K	5	9287	2786	30%

After the five rounds of the dataset augmentation process, we collect a security patch dataset of 12,073 instances, where 4076 ones belong to the NVD-based dataset and 7997 ones belong to the wild-based dataset. We also get a cleaned non-security patch dataset of 23,742 instances.

B. Performance of Nearest Link Search Method (RQ2)

To evaluate the performance of our nearest link search method, we compare it with three other data augmentation methods including:

- *Brute force search*: directly screening security patches from the wild.
- *Pseudo labeling* [19]: locating candidates from prediction results of single machine learning model with the highest confidence.
- *Uncertainty-based labeling* [28]: locating candidates from prediction results of multiple machine learning classifiers with the highest certainty (i.e., consensus).

Table III summarizes comparative evaluation results (i.e., the percentage of security patches and the confidence interval) under the 95% confidence level. Given the same training dataset (i.e., the NVD-based dataset with 4076 security patches and a non-security patch dataset of 8352 instances), we compare the performance of these four methods on recognizing security patch candidates from an unlabeled dataset of 200K random GitHub commits. For the brute force search, instead

TABLE III: Comparison with other augmentation methods.

Methods	Unlabeled Patches	Candidates	Security Patches* (%)
Brute Force Search	200K	200K	8(±1.7)%
Pseudo Labeling		4K	13(±1.8)%
Uncertainty-based Labeling		1K	12%
Nearest Link Search (ours)		4K	**29(±2.4)%**

* Sampled results based on 1K of candidates and 95% confidence level

TABLE IV: Performance w/o or w/ synthetic patches.

Dataset	Synthetic Dataset	Precision	Recall
NVD	-	82.1%	84.8%
NVD	17K Sec. + 20K NonSec.	86.0% **(+3.9%)**	87.2% **(+2.4%)**
NVD+Wild	-	92.9%	61.1%
NVD+Wild	58K Sec. + 129K NonSec.	93.0% (+0.1%)	61.2% (+0.1%)

Sec. = security patch; NonSec. = non-security patch

of manually verifying each of the 200K instances, we verify a random subset of 1K instances. The percentage of security patches is around 8%. For the pseudo labeling, we use the NVD-based dataset to train a model with the same features adopted in our nearest link search method. Among popular machine learning algorithms, we choose the Random Forest classifier that performs the best to rank the security patch candidates according to their confidence values and then select the top 4076 candidates. In the 1K subset of top candidates, about 13% of them are manually verified as security patches.

When conducting the uncertainty-based labeling, we implement ten classifiers including Random Forest, Support Vector Machine (SVM), Logistic Regression, Stochastic Gradient Descent (SGD) classifier, Sequential Minimal Optimization (SMO) classifier, Naive Bayes, Bayesian Network, J48 Decision Tree, Reduced Error Pruning Tree (REPTree), and Voted Perceptron using Weka [14]. An unlabeled GitHub commit is regarded as a candidate only if all ten classifiers predict it as a security patch. When applying the uncertain-based ensemble model, 1174 instances out of the 200K unlabeled data are predicted as security patches by all ten classifiers. Our manual checks show that 12% of 1174 instances are security patches.

For our nearest link search method, we manually verify 1K instances from 4076 candidates. We find around 29% instances are security patches. We further investigate the reasons that our method outperforms both the pseudo labeling method and the uncertainty-based labeling method. The main reason is that the distribution of security patches in the wild is different from that in the NVD-based dataset, which may be biased to certain types of vulnerabilities or popular software [20], [32]. Therefore, the models trained by the NVD-based dataset would not be able to well profile patches in the wild. In contrast, our nearest link search mainly targets at local distribution, i.e., finding the nearest neighboring instances of the existing dataset. Therefore, it has more tolerance on the difference between the NVD-based dataset and the wild dataset.

With the same amount of human labor (e.g., manually checking 1K subset), our experiments show that our nearest link search can help identify more security patches. Since there are hundreds of million repositories on GitHub, the number of unlabeled patches (i.e., commits) will be huge. Therefore, it is promising to construct an even larger wild-based security patch dataset by repeating the data augmentation process with more human efforts.

C. Evaluation of Synthetic Security Patches (RQ3)

To figure out if synthetic patches are useful and in which condition they are useful, we apply our oversampling technique on the NVD-based dataset and the wild-based dataset, respectively. For the NVD-based dataset, we create 16,836 artificial security patches and 19,936 artificial non-security ones. For the wild-based dataset, we generate a synthetic dataset containing 57,724 security patches and 128,736 non-security ones. We apply them into a task of automatic security patch identification and evaluate if the performance can be improved by including the corresponding synthetic dataset.

When only employing the NVD-based dataset, we randomly select 80% as the training set and use the remaining 20% as the testing set. When applying the NVD-based dataset and its synthetic dataset, we add all the synthetic data to the previous training set to train the model and conduct inference on the previous testing set for a fair comparison. We follow the same way to split and allocate the data for the *NVD+wild* dataset itself and the *NVD+wild* dataset along with its synthetic dataset. Note that our synthetic patches are generated solely based on the training set in each experiment. In our evaluation, we adopt the recurrent neural network (RNN) algorithm [30], which considers the source code of a given patch as a list of tokens including keywords, identifiers, operators, etc. In the RNN model, the current state depends on the current inputs and the previous state so that the model can learn the context information from tokens.

Comparative results are shown in Table IV. When solely depending on the NVD-based dataset, the classification precision of the RNN model is 82.1% with a recall of 84.8%. After adding the synthetic data into the NVD-based dataset, the classification precision increases by 3.9% (to 86.0%), and the F1 score can increase by 2.4% (to 87.2%). However, for the natural dataset containing both the NVD-based dataset and the wild-based dataset, we do not observe obvious improvement after adding the synthetic data. The model trained with the natural dataset only achieves 92.9% precision with the recall of 61.1%. After adding the synthetic dataset, there is only a slight increase, i.e., 0.1% in precision and 0.1% in recall. Note that recall of the last two rows are lower than the first two rows since the *NVD+wild* test dataset involves wild patches. We also try some traditional oversampling techniques like SMOTE and do not observe obvious performance increase.

The experimental results show that our oversampling technique is effective in the security patch identification task if we only have a small dataset (i.e., the NVD-based dataset). When we have a larger dataset (i.e., the combination of the NVD-based dataset and wild-based dataset), the synthetic data cannot lead to distinct improvement. The results are reasonable. Since the small dataset contains a limited number and patterns

TABLE V: Security patch distribution in PatchDB.

ID	Type of patch pattern	%*
1	add or change bound checks	10.8%
2	add or change null checks	9.1%
3	add or change other sanity checks	18.0%
4	change variable definitions	4.8%
5	change variable values	9.1%
6	change function declarations	1.8%
7	change function parameters	2.6%
8	add or change function calls	24.4%
9	add or change jump statements	1.7%
10	move statements without modification	5.0%
11	add or change functions (redesign)	12.0%
12	others	0.8%

* Sampled results based on 1K patches.

of patches, it cannot fully represent the feature space. In this case, the synthetic patches increase the number of control flow complexity as well as enrich the feature representations. As a result, the oversampling technique can boost the generalization ability of the trained model. In contrast, a larger security patch dataset may have already contained enough patch samples of various patterns so that the synthesis technique can only provide a marginal increase.

D. Dataset Composition (RQ4)

To figure out the composition of the PatchDB, we analyze it from the perspective of patch patterns. According to the definition of patch patterns in previous studies [35], [38], [41], we classify these security patches into 12 categories in Table V in terms of their code changes. The first three types that add sanity checks are common in the security patches since they directly block unsafe inputs. Given the fact that the bound and NULL pointer are the most frequent items in the sanity check, we consider them separately. Type 4 includes changing the data type from `int` to `unsigned int`, resizing a buffer, etc. Type 5 changes variable values, e.g., initialize memory to zero for preventing information leak. Type 6, 7, and 8 are related to fixing vulnerable functions and their parameters. Among them, Type 8 is the most common one (e.g., replacing an unsafe C library function `strcpy` with `strlcpy`, adding the lock and unlock before and after a raced operation, and calling release functions to avoid information leak). Type 9 adds or modifies the jump statements for the vulnerabilities that lack proper error handling. Type 10 moves some statements from one place to another with little or no modifications. Such fixes are usually for uninitialized use, use-after-free, etc. Type 11 rewrites the function logic with lots of different program changes. Type 12 refers to some uncommon minor changes that cannot be categorized into any of the above types.

From Table V, we can find that Type 8 is the most frequent class in the PatchDB. Type 1, 3, and 8 (i.e., several kinds of sanity checks and function call modifications) compose more than half of the PatchDB. Intuitively, given a security patch in the NVD-based dataset, the nearest link search method locates its nearest instance in the feature space as the candidate for dataset augmentation. Therefore, we wonder if the nearest link search changes the type distribution. In other words, facilitated

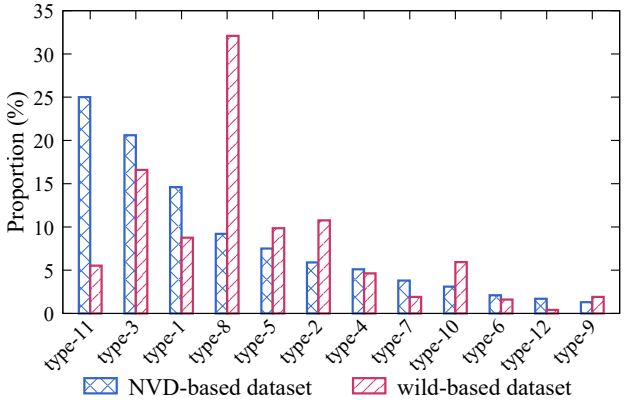

Fig. 6: Distribution comparison between NVD-based and wild-based datasets in terms of code changes.

with the nearest link search, is the type distribution of the wild-based dataset the same as that of the NVD-based dataset?

To answer this question, we manually classify a random subset according to the patch patterns for the NVD-based and wild-based dataset, respectively. The results in Figure 6 show that the wild-based dataset identified by the nearest link search differs from the original NVD-based dataset with regards to the type distribution. The type allocation of the NVD-based dataset conforms to a long tail distribution [7], where 3 out of the 12 types consist of around 60% of the NVD-based dataset, and most of the other 9 types are under 5%. In contrast, the type distribution of the wild-based dataset is largely different from the NVD-based dataset. Previous head class Type 11 only accounts for around 5% of the wild-based dataset. Type 8 becomes the first head class. Also, the ranks of previous tail classes are mostly changed. Thus, generated by the nearest link search, the wild-based dataset exhibits a different type distribution from the NVD-based dataset.

Although each security patch located by the nearest link search is the nearest neighbor of a security patch in the NVD-based dataset, their similarity may only be in the feature space, and these features are not one-to-one corresponding with security patch types. Between security and non-security patches, given a security patch, the nearest link search finds a similar instance that is also a security patch; however, they may not belong to the same security patch type. Therefore, the wild-based dataset identified by the nearest link search can have a dissimilar distribution from the NVD-based dataset. Meanwhile, such differences bring some benefits. The main problem of the long tail distribution is that there is not enough data for tail classes. In that case, machine learning would not perform well when handling those minority instances. The wild-based dataset solves this problem to a certain extent by introducing more varieties to the PatchDB.

During this analysis, we also manually classify a total number of 5K security patches in PatchDB into these patch patterns. We will also make these materials public for further research. More potential use will be discussed in Section V.

TABLE VI: Impacts of datasets over learning-based models

Training Dataset	Algorithm	Test Dataset	Precision	Recall
NVD	Random Forest	NVD	58.4%	21.7%
		Wild	58.0%	19.5%
	RNN	NVD	82.8%	83.2%
		Wild	88.3%	24.2%
NVD+Wild	Random Forest	NVD	90.1%	22.5%
		Wild	91.8%	44.6%
	RNN	NVD	92.8%	60.2%
		Wild	92.3%	63.2%

E. Quality of the PatchDB (RQ5)

To evaluate the quality of our collected dataset, we employ PatchDB in the task of automatic security patch identification. More specifically, we use both the NVD-based and wild-based dataset to train a learning-based model that identifies if a given patch is security-related. Then, we compare its performance with the one trained by only the NVD-based dataset. In our experiments, two machine learning models are implemented. One is the random forest classifier with the statistical features of patches, and the other is the recurrent neural network classifier that can extract context information from tokens.

The experimental setup and results are shown in Table VI. For the NVD-based dataset, we randomly choose 80% instances as the training set and the remaining 20% as the test set. The wild-based dataset is split in the same way. Then, for a fair comparison, we combine the training set of both the NVD-based and wild-based dataset as the training data. When using the NVD-based test dataset, the Random Forest (and RNN) model trained with the NVD-based dataset can achieve 58.4% (and 82.8%) precision with 21.7% (and 83.2%) recall. The testing precision and recall would be 90.1% (and 92.8%) as well as 22.5% (and 60.2%) recall if we train the model with both NVD-based dataset and wild-based dataset. The training dataset has little impact on the model when tested on the NVD-based dataset. However, when using the wild-based test dataset, the precision and recall drop to 58.0% (and 88.3%) and 19.5% (and 24.2%) if the random forest (and RNN) model is trained with the NVD-based dataset. Therefore, the machine learning model trained by the NVD-based dataset exhibits insufficient generalization ability due to the limited number and patterns of instances. In contrast, the models trained with both the NVD-based dataset and the wild-based dataset have better generalization ability. The performance remains stable no matter which testing dataset is used, proving that the model can be applied to unknown patch samples.

The differences between classification models are also demonstrated in Table VI. With the same training data and testing data, the RNN model has a better performance than the Random Forest model. Compared with the statistic syntactic features (Table I), the RNN model can also seize the context information between programming tokens, which provides valuable insight into the programming language processing.

V. Discussion

We describe several usage scenarios of PatchDB and how the dataset may promote the related research and applications. We also discuss some limitations as well as future work.

A. Usage Scenarios of PatchDB

1) Vulnerability/Patch Presence Detection: Since security patches comprise both the vulnerable code and corresponding fixes, they can be used to detect vulnerable code clone by using patch-enhanced vulnerability signatures [9], [36]. Such works generate signatures directly from the code gadgets. Hence, more security patch instances enable more vulnerability signatures for matching and thus enhances the detection capability. From another perspective, patching status is critical for downstream software, which motivates the need for reliable patch presence testing. The PatchDB identifies 8K silent security patches that are not provided in the NVD. The presence of such patches can be tested in the downstream software [17], [40]. Also, a binary security patch dataset could be constructed by compiling the source code in our dataset.

2) Automatic Patch Generation: Since previous patch analysis works are conducted on a small dataset, they are constrained to summarize the fix patterns of some common patch types, e.g., sanity testing. The main reason is that they lack enough instances to perform their study. In contrast, our analysis on the PatchDB in terms of code changes (Section IV-B-2) presents that there are still many security patches with multiple fix patterns so that our large-scale patch dataset could be used to summarize more patch patterns. In Table VII, we show two examples of fix patterns concluded by ourselves based on observation of the PatchDB that have never been studied by previous study [24], [35], [38], i.e., race condition and data leakage. These patterns describe how security patches fix the corresponding security impacts caused by vulnerable operations. For the race condition, the patches typically add and release lock to guarantee the atomicity for a vulnerable operation. For the data leakage, the patches often release the critical value after the last normal operation to avoid further vulnerable operation. With a large-scale security patch dataset, more complex patch patterns can be discovered so that semantics can be learned for automatically generating more types of security patches.

TABLE VII: Example of fix patterns

Race Condition	Data Leakage

`...`
`+ lock(CV);`
`vulnerable_op(CV, ...);`
`+ unlock(CV);`
`...`
	`normal_op(CV);`
	`...`
	`+ release(CV);`
	`...`
	`vulnerable_op(CV, ...);`

CV = critical value

3) Benchmark: The PatchDB can also be used as a benchmark. Since the PatchDB is the largest-scale dataset of security patches to the best of our knowledge, it is closer to the practical scenario and enlarges the spectrum of the evaluation. Also, since it is collected from 313 GitHub repositories other than some specific projects, it provides a good benchmark to test the generalization capability of target techniques.

B. Limitations and Future Work

Our work currently focuses on C/C++ languages that are with the highest number of vulnerabilities [34]. The syntactic features identified in Table I may be commonly shared by the patches for different languages (e.g., changes of if statements, loops, and logical operators). Therefore, our system could be extended to other programming languages by customizing their syntax parsing related features. However, for safe languages like Rust that have much fewer vulnerabilities, it may be difficult to collect a large-size security patch dataset. We leave the extensions to other languages as future work.

Similar to previous work [20], we assume that all the information retrieved from the NVD is correct. That is to say, we assume the patches crawled from the URLs provided by the NVD are for the corresponding CVE entry. However, we observe up to 1% of patches may not be correct. For example, the provided link is for a brand new version that mingles multiple code differences where security fix is part of that. We consider the proportion of incorrect patches is small enough to be ignored in our analysis. Also, the NVD may be biased towards certain types of software. Given the wide range of software included by the NVD, we argue that it will remain largely applicable for most open source software.

VI. Related Works

Patch Datasets. Since a security patch aggregates both vulnerable code and the corresponding modifications at the same time, many vulnerability detection research constructs security patch datasets. Kim et al. [18] acquire security patches from eight well-known Git repositories to detect vulnerable code clone. Z. Li et al. [21] build a Vulnerability Patch Database (VPD) that consists of 19 products. However, the size of these datasets is not sufficient to perform a machine learning-based study and may introduce biases to analysis results. Although SARD provides some samples that mitigate the vulnerabilities, it mainly focuses on vulnerable code and most of the samples are artificial. By querying thousands of CVE records for open source projects on the NVD, F. Li et al. [20] build a large-scale security patch database. Further, considering silent security patches, Xiao et al. [36] enrich the dataset with commits obtained from their industrial collaborator. However, such datasets are not publicly accessible.

Besides, there are several web-based patch or bug tracking systems. Patchwork [6], a patch management system, catches patches sent to the mailing list, but it is mainly used for several Linux kernel subsystems. Bug tracking systems like Bugzilla [1] may provide patch information in corresponding reports. Yet not all the bug reports contain such information and they do not distinguish between security and non-security patches. These limitations motivate us to construct a large dataset of security patches from various types of projects.

Patch Analysis. Recently, there is an increasing number of works on patch analysis. Most of them focus on investigating the textual information (e.g., bug report, commit message, etc.), which does not require the retrieval and analysis of the source code. They use supervised and unsupervised learning techniques to classify patches [13], [16], [43]. However, they cannot handle the situation where the documentation of security patches is inaccurate or even totally missing due to different maintainers, limited security domain knowledge, and changing regulations during the software life cycle.

At the source code level, Zhong et al. [42] conduct an empirical study on bug fixes from six popular JAVA projects. Soto et al. [29] focus on patterns, replacements, deletions, and additions of bug fixes. Perl et al. [27] study the attributes of commits that are more likely to introduce vulnerabilities. Machiry et al. [25] analyze safe patches that do not disrupt the intended functionality of the program. However, all these works do not distinguish security patches from normal bug fixes. Zaman et al. [39] discover the differences between security patches and performance bugs on a specific project - Mozilla Firefox. Li et al. [20] are the first one to perform a large-scale empirical study of security patches versus non-security bug fixes, discussing the metadata characteristics and life cycles of security patches.

Some studies utilize machine learning-based models to identify the type of a given patch [31]–[33], while the deficiency of patch instances restricts the application of the robust classifier (e.g., deep learning model). Also, most of these models are trained with a dataset from single or multiple software projects, which provide limited generalization capacity in the wild. In contrast, our work provides a large dataset from over 300 GitHub repositories and we use a new oversampling method to further increase the variants at the source code level. At the binary level, Xu et al. [37] present a scalable approach to identify the existence of a security patch through semantic analysis of execution traces. With the help of signatures generated from open-source patches, some methods [17], [40] test if the target binaries have been patched.

VII. Conclusion

In this work, we construct a large-scale dataset of security patches called PatchDB. In particular, we develop a novel nearest link search approach to help locate the most promising candidates of security patches from an unlabeled dataset, reducing the workload of the manual verification. Also, we propose a new oversampling method to synthesize patches at the source code level, which is effective to increase the variance of the patch dataset. We conduct a set of experiments to study the composition and quality of PatchDB and verify the effectiveness of our proposed algorithms. The results of a comprehensive evaluation show that PatchDB is promising to facilitate the patch analysis and vulnerability detection techniques.

VIII. Acknowledgments

This work was partially supported by the US Department of the Army grant W56KGU-20-C-0008, the Office of Naval Research grants N00014-18-2893, N00014-16-1-3214, and N00014-20-1-2407, and the National Science Foundation grants CNS-1815650 and CNS-1822094.

REFERENCES

[1] Bugzilla. https://www.bugzilla.org.

[2] Common Vulnerabilities and Exposures. https://cve.mitre.org.

[3] GitHub. https://github.com.

[4] LLVM. https://llvm.org.

[5] National Vulnerability Database. https://nvd.nist.gov.

[6] Patchwork. http://patchwork.ozlabs.org.

[7] Chris Anderson and Mia Poletto Andersson. Long tail. 2004.

[8] Paul E Black. Sard: Thousands of reference programs for software assurance. *J. Cyber Secur. Inf. Syst. Tools Test. Tech. Assur. Softw. Dod Softw. Assur. Community Pract*, 2(5), 2017.

[9] Benjamin Bowman and H Howie Huang. VGraph: A Robust Vulnerable Code Clone Detection System Using Code Property Triplets. In *2020 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 53–69. IEEE, 2020.

[10] Center for Assured Software National Security Agency. Juliet Test Suite v1.2 for C/C++ User Guide. https://samate.nist.gov/SRD/resources/Juliet_Test_Suite_v1.2_for_C_Cpp_-_User_Guide.pdf, 2018.

[11] Nitesh V Chawla, Kevin W Bowyer, Lawrence O Hall, and W Philip Kegelmeyer. Smote: synthetic minority over-sampling technique. *Journal of artificial intelligence research*, 16:321–357, 2002.

[12] Hong Cui, Jingjing Zhang, Chunfeng Cui, and Qinyu Chen. Solving large-scale assignment problems by kuhn-munkres algorithm. 2016.

[13] Dipok Chandra Das and Md Rayhanur Rahman. Security and performance bug reports identification with class-imbalance sampling and feature selection. In *2018 Joint 7th International Conference on Informatics, Electronics & Vision (ICIEV) and 2018 2nd International Conference on Imaging, Vision & Pattern Recognition (icIVPR)*, pages 316–321. IEEE, 2018.

[14] E. Frank, M. A. Hall, G. Holmes, R. Kirkby, B. Pfahringer, and I. H. Witten. *Weka: A machine learning workbench for data mining.*, pages 1305–1314. Springer, Berlin, 2005.

[15] Michael Gegick, Pete Rotella, and Tao Xie. Identifying security bug reports via text mining: An industrial case study. In *2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)*, pages 11–20. IEEE, 2010.

[16] Katerina Goseva-Popstojanova and Jacob Tyo. Identification of security related bug reports via text mining using supervised and unsupervised classification. In *2018 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, pages 344–355. IEEE, 2018.

[17] Zheyue Jiang, Yuan Zhang, Jun Xu, Qi Wen, Zhenghe Wang, Xiaohan Zhang, Xinyu Xing, Min Yang, and Zhemin Yang. Pdiff: Semantic-based patch presence testing for downstream kernels. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 1149–1163, 2020.

[18] Seulbae Kim, Seunghoon Woo, Heejo Lee, and Hakjoo Oh. Vuddy: A scalable approach for vulnerable code clone discovery. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 595–614. IEEE, 2017.

[19] Dong-Hyun Lee. Pseudo-label: The simple and efficient semi-supervised learning method for deep neural networks. In *Workshop on challenges in representation learning, ICML*, volume 3, 2013.

[20] Frank Li and Vern Paxson. A large-scale empirical study of security patches. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2201–2215, 2017.

[21] Zhen Li, Deqing Zou, Shouhuai Xu, Hai Jin, Hanchao Qi, and Jie Hu. Vulpecker: an automated vulnerability detection system based on code similarity analysis. In *Proceedings of the 32nd Annual Conference on Computer Security Applications*, pages 201–213, 2016.

[22] Zhen Li, Deqing Zou, Shouhuai Xu, Xinyu Ou, Hai Jin, Sujuan Wang, Zhijun Deng, and Yuyi Zhong. Vuldeepecker: A deep learning-based system for vulnerability detection. *arXiv preprint arXiv:1801.01681*, 2018.

[23] Guanjun Lin, Wei Xiao, Jun Zhang, and Yang Xiang. Deep learning-based vulnerable function detection: A benchmark. In *International Conference on Information and Communications Security*, pages 219–232. Springer, 2019.

[24] Kangjie Lu, Aditya Pakki, and Qiushi Wu. Detecting missing-check bugs via semantic-and context-aware criticalness and constraints inferences. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1769–1786, 2019.

[25] Aravind Machiry, Nilo Redini, Eric Camellini, Christopher Kruegel, and Giovanni Vigna. Spider: Enabling fast patch propagation in related software repositories. In *2020 IEEE Symposium on Security and Privacy (SP). IEEE*, 2020.

[26] Masao Ohira, Yutaro Kashiwa, Yosuke Yamatani, Hayato Yoshiyuki, Yoshiya Maeda, Nachai Limsettho, Keisuke Fujino, Hideaki Hata, Akinori Ihara, and Kenichi Matsumoto. A dataset of high impact bugs: Manually-classified issue reports. In *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*, pages 518–521. IEEE, 2015.

[27] Henning Perl, Sergej Dechand, Matthew Smith, Daniel Arp, Fabian Yamaguchi, Konrad Rieck, Sascha Fahl, and Yasemin Acar. VCCFinder: finding potential vulnerabilities in open-source projects to assist code audits. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 426–437. ACM, 2015.

[28] Richard Segal, Ted Markowitz, and William Arnold. Fast uncertainty sampling for labeling large e-mail corpora. In *CEAS*. Citeseer, 2006.

[29] Mauricio Soto, Ferdian Thung, Chu-Pan Wong, Claire Le Goues, and David Lo. A deeper look into bug fixes: patterns, replacements, deletions, and additions. In *2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)*, pages 512–515. IEEE, 2016.

[30] Jinsong Su, Zhixing Tan, Deyi Xiong, Rongrong Ji, Xiaodong Shi, and Yang Liu. Lattice-based recurrent neural network encoders for neural machine translation. In *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence*, AAAI'17, page 3302–3308. AAAI Press, 2017.

[31] Yuan Tian, Julia Lawall, and David Lo. Identifying linux bug fixing patches. In *2012 34th international conference on software engineering (ICSE)*, pages 386–396. IEEE, 2012.

[32] Xinda Wang, Kun Sun, Archer Batcheller, and Sushil Jajodia. Detecting" 0-day" vulnerability: An empirical study of secret security patch in oss. In *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 485–492. IEEE, 2019.

[33] Xinda Wang, Shu Wang, Kun Sun, Archer Batcheller, and Sushil Jajodia. A machine learning approach to classify security patches into vulnerability types. In *2020 IEEE Conference on Communications and Network Security (CNS)*, pages 1–9. IEEE, 2020.

[34] White Source Software. What are the most secure programming languages? https://www.whitesourcesoftware.com/most-secure-programming-languages/.

[35] Qiushi Wu, Yang He, Stephen McCamant, and Kangjie Lu. Precisely characterizing security impact in a flood of patches via symbolic rule comparison. In *Proceedings of the 27th Annual Network and Distributed System Security Symposium (NDSS'20)*, 2020.

[36] Yang Xiao, Bihuan Chen, Chendong Yu, Zhengzi Xu, Zimu Yuan, Feng Li, Binghong Liu, Yang Liu, Wei Huo, Wei Zou, et al. MVP: Detecting vulnerabilities using patch-enhanced vulnerability signatures. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 1165–1182, 2020.

[37] Zhengzi Xu, Bihuan Chen, Mahinthan Chandramohan, Yang Liu, and Fu Song. Spain: security patch analysis for binaries towards understanding the pain and pills. In *Proceedings of the 39th International Conference on Software Engineering*, pages 462–472. IEEE Press, 2017.

[38] Zhengzi Xu, Yulong Zhang, Longri Zheng, Liangzhao Xia, Chenfu Bao, Zhi Wang, and Yang Liu. Automatic hot patch generation for android kernels. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 2397–2414, 2020.

[39] Shahed Zaman, Bram Adams, and Ahmed E Hassan. Security versus performance bugs: a case study on firefox. In *Proceedings of the 8th working conference on mining software repositories*, pages 93–102, 2011.

[40] Hang Zhang and Zhiyun Qian. Precise and accurate patch presence test for binaries. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 887–902, 2018.

[41] Lei Zhao, Yuncong Zhu, Jiang Ming, Yichen Zhang, Haotian Zhang, and Heng Yin. Patchscope: Memory object centric patch diffing. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 149–165, 2020.

[42] Hao Zhong and Zhendong Su. An empirical study on real bug fixes. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 913–923. IEEE, 2015.

[43] Yaqin Zhou and Asankhaya Sharma. Automated identification of security issues from commit messages and bug reports. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 914–919, 2017.