Locating the Security Patches for Disclosed OSS Vulnerabilities with Vulnerability-Commit Correlation Ranking

Xin Tan*
Fudan University
18212010028@fudan.edu.cn

Yuan Zhang*
Fudan University
yuanxzhang@fudan.edu.cn

Chenyuan Mi
Fudan University
20210240143@fudan.edu.cn

Jiajun Cao
Fudan University
20210240046@fudan.edu.cn

Kun Sun
George Mason University
ksun3@gmu.edu

Yifan Lin
Fudan University
19210240159@fudan.edu.cn

Min Yang
Fudan University
m_yang@fudan.edu.cn

ABSTRACT

Security patches play an important role in defending against the security threats brought by the increasing OSS vulnerabilities. However, the collection of security patches still remains a challenging problem. Existing works mainly adopt a matching-based design that uses auxiliary information in CVE/NVD to reduce the search scope of patch commits. However, our preliminary study shows that these approaches can only cover a small part of disclosed OSS vulnerabilities (about 12%-53%) even with manual assistance.

To facilitate the collection of OSS security patches, this paper proposes a ranking-based approach, named PatchScout, which ranks the code commits in the OSS code repository based on their correlations to a given vulnerability. By exploiting the broad correlations between a vulnerability and code commits, patch commits are expected to be put to front positions in the ranked results. Compared with existing works, our approach could help to locate more security patches and meet a balance between the patch coverage and the manual efforts involved. We evaluate PatchScout with 685 OSS CVEs and the results show that it helps to locate 92.70% patches with acceptable manual workload. To further demonstrate the utility of PatchScout, we perform a study on 5 popular OSS projects and 225 CVEs to understand the patch deployment practice across branches, and we obtain many new findings.

1 INTRODUCTION

With the increase of open source software (OSS), the number of reported OSS vulnerabilities has also experienced a rapid growth. As reported by WhiteSource [2], the number of disclosed OSS vulnerabilities in 2019 skyrocketed to over 6,000, which rose by nearly 50% compared to 2018. To mitigate these vulnerabilities, developers usually resort to security patches.

In practice, security patches are central in defending against security threats. First, the patches can be directly applied to fix the corresponding vulnerabilities. Second, hot-patches can be derived from the original security patches to ease the deployment of security patches, with the help of hot-patching frameworks [13, 15, 24, 51]. Third, security patches are useful in facilitating downstream tasks, such as vulnerable code clone detection [37, 42, 46, 71, 72], patch presence testing [27, 31, 38, 77]. Finally, due to their importance, security patches have become an important target to study, such as understanding their development process and complexities [44, 63, 78] and their perceptions by end-users [61]. Therefore, the collection of security vulnerabilities, as well as their corresponding patches, become an important asset for the community.

CVE [25] and NVD [54] are two popular public references for security vulnerabilities. In particular, CVE gives every reported vulnerability an identification number (called CVE-ID), a description, and at least one public reference. Further, NVD incorporates all vulnerabilities in CVE with enhanced vulnerability information such as severity scores (CVSS), vulnerability type (CWE), and affected software configurations (CPE). Based on these vulnerability databases, existing and emerging security vulnerabilities have been efficiently shared with interested users; however, how to accurately locate their security patches still remains an open problem.

In essence, security patches are code commits that are developed/deployed by OSS developers in their code repositories. However, due to the large number of code commits in a code repository (e.g., there are about 936k commits in Linux kernel till version 5.8), locating security patches is quite laborious. To reduce the search scope, existing works usually turn to extra auxiliary information in the vulnerability database. For example, Perl *et al.* [56] and Kim *et al.* [42] locate security patches from the code commits that mention the corresponding CVE-ID; other works [10, 44, 56, 68] locate security patches from the external reference URLs in the CVE/NVD pages. As it will be demonstrated in our preliminary study (see §2), these approaches only cover a small part of disclosed vulnerabilities due to two reasons: 1) only few vulnerability information in CVE/NVD is used to reduce the search scope, which is usually incomplete (and sometimes incorrect); 2) these approaches are matching-based, which means they either give few candidates when matched or give no results when unmatched.

In the field of mining software repositories (MSR), there is a line of research that aims to predict the components (e.g., files) to be fixed for a bug [12, 36, 41, 43, 45, 60, 67, 69, 74, 76, 79]. By mining the correlations between a bug and different software components, these works rank the software components to reflect their likelihood

*co-first authors

CCS '21, November 14–19, 2021, Virtual Event, South Koera
2021. ACM ISBN 978-1-4503-7089-9/20/11…$15.00
https://doi.org/10.1145/3372297.3417240

of being fixed for this bug. Inspired by these works, we propose to transform the search problem of locating security patches into a ranking problem on code commits, and we design a system called PatchScout to incorporate this idea. For code commit ranking, PatchScout features a new technique, called *vulnerability-commit correlation ranking*, which exploits the broad correlations between the vulnerability and the code commits to put more relevant code commits to the front. Different from existing approaches, PatchScout leverages more types of vulnerability information to estimate the relevance of a code commit to a given vulnerability. In general, four groups of correlation features between a code commit and a vulnerability are considered, namely, *vulnerability identifier*, *vulnerability location*, *vulnerability type*, and *vulnerability descriptive texts*. Based on these features, PatchScout further trains a RankNet [20] model to rank code commits. Since patch commits share a lot of relevant information with the vulnerability, they are expected to be put to front positions in the ranked results. With the code commits ranked, the efforts to locate security patches are dramatically reduced than those exploring all code commits.

Compared with existing approaches, PatchScout has several advantages. First, it uses a wide range of vulnerability information, so it can tolerate more on the incomplete/incorrect vulnerability information in CVE/NVD pages. Second, the ranking-based solution has a higher chance to extract security patches than the existing matching-based solutions since it always gives ranked results while existing solutions only provide results when matched. Third, by exploiting the underlying connections (even weak ones) between a vulnerability and code commits, the *vulnerability-commit correlation ranking* mechanism in PatchScout is capable of locating patches for more vulnerabilities. Similar to existing works, PatchScout also requires manual efforts to finally locate the security patches from the ranked code commits; however, it can be used as a search engine to facilitate the locating of security patches from a large number of code commits. Without PatchScout, security experts may need to explore thousands of commits to locate the security patch(es) for one vulnerability. With PatchScout, since the commits are ranked, only a few high-ranked commits are expected to be checked before the security patches are located.

We evaluate the effectiveness of PatchScout on locating security patches for real-world OSS vulnerabilities. Specifically, we train a ranking model for PatchScout with 943 CVEs and their patches, and test its performance with other 685 CVEs. The results show that PatchScout successfully ranks the patch commit for 69.49% CVEs at the first position among all the code commits and helps to locate the security patches for 92.70% CVEs at the cost of checking 4.32 commits on average. Compared with existing approaches that can locate the patches for at most 47.59% CVEs and require to check 1.83 commits per CVE, PatchScout can locate 85.40% patches with almost the same amount of manual work involved. Our evaluation shows that PatchScout effectively facilitates the locating of security patches by delicately balancing the coverage of the security patches and the manual efforts involved.

To further illustrate the security benefits of PatchScout, we apply it to conduct a study on the patch deployment practice across branches. Our study collects 225 CVEs from 5 OSS projects and uses PatchScout to locate the security patches for these vulnerabilities on 83 branches. With the help of PatchScout, we successfully locate 1,985 patches while existing approaches can only locate 1,087 patches. Our study discovers that a large portion of branches are vulnerable and still stay unpatched, and some patched branches suffer from a quite long patch lag. Besides, we find that 38 CVEs miss the reporting of 152 affected versions in CVE/NVD. CVE maintainers have confirmed our findings and updated their database accordingly. In addition, our study gives a landscape about the different types of patch backporting situations across branches and analyzes different levels of technical difficulties in patch backporting, which raises some new research problems.

In summary, we make the following contributions.

- We propose a new idea of locating security patches for disclosed OSS vulnerabilities, which transforms the search problem of patch commits into a ranking problem.
- We present a new technique, i.e. *vulnerability commit correlation ranking*, which ranks code commits based on their relevance to a vulnerability from multiple correlation features.
- We evaluate the effectiveness of our proposed approach, and the results show that our approach significantly outperforms existing ones in both patch coverage and required manual efforts.
- We conduct the first study on the patch deployment practice across branches, which draws many interesting findings and concludes several important research opportunities.

2 PRELIMINARY STUDY

Security patches play a central role in defending against security vulnerabilities; however, many vulnerabilities are disclosed without their patches being published at the same time. Moreover, locating patches from tens of thousands of code commits is an extremely time-consuming and laborious job. To avoid searching the patches in a large scope, researchers usually leverage some extra references [10, 44, 56, 68] or keywords [42, 56] in CVE/NVD databases.

From existing works, we identify three methods that can be used to locate the security patches of publicly reported vulnerabilities.

- *M1: searching commit messages with CVE-ID.* Sometimes, developers may declare the CVE-ID in the commit message of the security patch, so the search scope of security patches can be reduced by using the CVE-ID as the keyword to filter those irrelevant code commits. Some previous works [42, 56] apply this method to collect patches for further research.
- *M2: checking commit-like URLs in CVE/NVD.* External resources that are related to each CVE are provided as reference links in CVE/NVD. Since security patches may also be collected as reference links, previous works [44, 56, 68] rely on identifying the reference URLs in a commit format[1] as patches.
- *M3: checking patch-tagged URLs in the NVD.* In addition to collecting reference links for each CVE, the NVD tags some links to indicate what type of resources they provide. In particular, the NVD assigns a "*Patch*" tag for a link that refers to a security patch[2]. Therefore, the patch-tagged URLs in NVD pages have been used [10] to identify the security patches.

Though the above approaches reduce the efforts in locating security patches, their performances have not been systematically explored. To further understand the difficulties in locating security

[1] e.g., https://gitlab.gnome.org/GNOME/libxml2/commit/0e1a49c89076
[2] e.g., https://nvd.nist.gov/vuln/detail/CVE-2015-1474

patches and the performance of existing approaches, we perform a preliminary study that consists of the following two experiments.

Experiment-1: How many security patches can be located by existing approaches? To conduct the experiment, we first need a vulnerability dataset. As reported by WhiteSource [1], more than half of disclosed vulnerabilities are C/C++ vulnerabilities. Therefore, our experiment chooses C/C++ vulnerabilities as the target. In particular, we take four steps to collect a large set of disclosed C/C++ OSS vulnerabilities and the code repositories of these OSS projects: 1) we crawl all the vulnerabilities reported between January 2015 and July 2020 from the NVD; 2) we select C/C++ vulnerabilities from them by using file suffix (e.g., .c, .cpp, .cxx, etc.) as the keywords to match the vulnerability description; 3) we combine keyword searching and manual inspection to keep only C/C++ OSS vulnerabilities and confirm their affected OSS projects; 4) we manually locate the code repository for each affected OSS project and automatically clone these repositories. In all, we collect 6,628 C/C++ vulnerabilities belonging to 798 OSS projects.

The above three approaches (aka M1, M2, and M3) are applied to locate the security patches of the collected OSS vulnerabilities. Since these approaches may report wrong patch commits, their results should be manually verified. To limit the manual efforts in verifying the results, we first count the number of CVEs that at least one candidate patch could be located using each of these methods. Note that this result measures the upper bound of the coverage for each method in locating security patches. As shown in Table 1, all three approaches have a low coverage. Searching commit messages with CVE-ID (M1) only covers 12.01% of the CVEs, which implies that most developers do not explicitly declare the fixed vulnerability in the patch commit. With the help of the manually-collected external URLs in CVE/NVD, checking commit-like URLs (M2) and checking path-tagged URLs (M3) achieve the coverage of 43.33% and 52.90%, respectively. Besides, even we combine all these three methods, we can only cover the patches for 66.57% of the CVEs. This experiment illustrates the patching information provided by CVE/NVD is incomplete, and existing approaches cannot locate security patches for a large part of the disclosed vulnerabilities.

Experiment-2: How many manual efforts are needed in confirming the security patches and what is the precision of these approaches? We randomly select 20% potential patches reported by each approach and manually verify them. Besides, from the potential patches that are covered by the combination method of M1, M2, and M3, we also randomly select 20% for manual confirmation. In all, it takes three security researchers 155 man-hours to verify these commits. To verify if an identified commit is a correct patch for the corresponding vulnerability, participators carefully examine the commit and the vulnerability information and refer to public materials (e.g., bug reports) when needed.

The detailed verification results are presented in Table 1. It turns out that though these approaches require users to check no more than 2 commits per CVE, incorrect patch commits are common in their results. For example, although M3 gives patch candidates for 52.90% CVEs, only 47.42% of them are correct patches. Besides, M1 and M2 were used by Perl *et al.* [56] and Li *et al.* [44] to collect security patches, because they found no false positives of the two approaches in a small test set which includes less than 100 results.

Table 1: The patches located for 6,628 CVEs.

Approach	Covered CVEs	AVG Commits to Check	Commits to Verify	Commits Verified as Patches
M1	796 (12.01%)	1.93	317	273
M2	2,872 (43.33%)	1.26	721	688
M3	3,506 (52.90%)	1.63	1,162	551
M1+M2+M3	4,412 (66.57%)	1.78	1,531	1,029

However, in our evaluation with a larger test set, we find that both approaches report incorrect patch results, rendering that existing works may suffer from using incorrect patches in their works. This finding also demonstrates that collecting large-scale security patches is necessary and challenging. Note the common reasons for the false positives of these methods are: i) non-patch commits (e.g. test cases) may mention CVE-ID/Bug-ID in the commit messages (M1); ii) commit-like URLs in NVD may relate to non-patch commits (M2); iii) NVD maintainers may incorrectly tag patches (M3).

Key Findings. Our preliminary study shows that all existing approaches suffer from a low coverage in locating security patches and require manual efforts to verify their results. We find two main limitations for existing approaches. First, they do not efficiently use the vulnerability information in CVE/NVD. When only leveraging few information (e.g., CVE-ID, reference URLs) that is usually incomplete and sometimes incorrect to identify patch commits, existing approaches cannot cover a large portion of security patches. Second, they all leverage matching-based approaches, which lead to either few matched commits or no results at all.

3 APPROACH OVERVIEW

As shown in §2, existing approaches cannot effectively locate the security patches for a large number of disclosed vulnerabilities. To address this problem, this section first illustrates our new idea of locating security patches, i.e. *vulnerability-commit correlation ranking*, and then describes the core features used by our ranking.

3.1 Key Idea

Goals. Before introducing our new approach of locating security patches, we first clarify the goals that our approach should meet.

- *Goal-1: Help to locate more patches.* As our preliminary study shows, existing works cannot cover a large number of vulnerabilities. Therefore, our primary goal is to cover more patches for better fighting against vulnerability threats and preparing for more representative patch studies [38, 44, 56, 77].
- *Goal-2: Limit the involved manual efforts.* It is difficult to determine if a security patch is the correct one for a specific vulnerability, except that patch developers explicitly state it. As a result, it usually requires manual efforts to verify the results. Our goal is to reduce the manual efforts involved in confirming the security patches. In other words, our approach targets locating more patches with the same amount of manual work.

Observations. We have the following observations that can help to achieve our goals.

Observation-1: security patches are usually found as code commits in the OSS code repositories. In essence, a security patch is written by the OSS developers or submitted by security experts. To fix the vulnerability, it should be merged into the code repository. Therefore, security patches can be usually found as code commits.

Off-by-one error in *epan/dissectors/packet-gsm_abis_oml.c* in the GSM A-bis OML dissector in Wireshark 1.12.x before 1.12.10 and 2.x before 2.0.2 allows remote attackers to cause a **denial of service** (**buffer over-read** and application crash) via a crafted packet that **triggers a 0xff tag value**.

(a) NVD Description of CVE-2016-4417

gsm_abis_oml: fix **buffer overrun**
Do not read outside boundaries **when tag is exactly 0xff**.

```
    tag = tvb_get_guint8(tvb, offset);
    tdef = find_tlv_tag(tag);
        ...
        return &nm_att_tlvdef_base.def[tag];
...
```

```
1  diff --git a/epan/dissectors/packet-gsm_abis_oml.c b/epan/dissectors/packet-gsm_abis_oml.c
2  index a6158c3..543b034 100644
3  --- a/epan/dissectors/packet-gsm_abis_oml.c
4  +++ b/epan/dissectors/packet-gsm_abis_oml.c
5  @@ -618,7 +618,7 @@ struct tlv_def {
6   };
7
8   struct tlv_definition {
9  -        struct tlv_def def[0xff];
10 +        struct tlv_def def[0x100];
11  };
12
13  enum abis_nm_ipacc_test_no {
```

(b) Patch Commit of CVE-2016-4417

Figure 1: A motivating example to illustrate the correlations between a vulnerability and its patch commit.

In other words, the patch for a security vulnerability can be located by exploring all code commits as long as the patch exists.

Observation-2: there are broad correlations between the vulnerability and its security patch commit. In a security patch commit, developers may refer to the fixed vulnerability in various ways, such as directly mentioning it, explaining how this commit fixes the vulnerability, and describing the impact of this commit. Meanwhile, the vulnerability information in CVE/NVD usually describes the vulnerability from many aspects, such as the vulnerability type, vulnerability location, vulnerability impact, etc. Thus, there exist broad correlations between the vulnerability and its patch, facilitating the locating of security patches for a given vulnerability.

Figure 1 gives an example to illustrate the correlations between the vulnerability description of CVE-2016-4417 [53] and its patch [70], including vulnerability location, vulnerability type, and descriptive texts. First, both the vulnerability description and the commit message mention the same vulnerability location (words in green). In addition, the code change location in the code diff is also consistent with the vulnerability location. Second, the vulnerability description introduces the type and impact (words in red) of the CVE, while the commit message also claims it fixes a "buffer overrun" bug. Besides, the code diff also indicates it fixes a buffer-overrun bug by updating the buffer size from 0xff to 0x100. Third, the vulnerability description describes some other characteristics of the vulnerability, such as how to trigger the vulnerability (words in blue), while the commit message also contains similar descriptive texts. Considering all these correlations, we can easily mark it as the security patch for CVE-2016-4417.

Observation-3: The correlations between a vulnerability and a code commit enable us to locate security patches that cannot be located by the matching-based solutions. Since existing solutions introduced in §2 adopt a matching-based approach that only gives results on exactly matching, they cannot locate the patch commit in the example of Figure 1, which does not specify the CVE-ID and is not specified in CVE/NVD. In contrast, we can locate it by exploiting the correlations between the vulnerability and the code commit.

New Approach. Based on the above observations, we propose a *vulnerability-commit correlation ranking* approach, which locates security patches by ranking all the code commits according to the correlation between one commit and the corresponding vulnerability information. Our approach works like a search engine, which finds and ranks all the pages (code commits) that are highly correlated to a given short description (vulnerability).

Compared to matching-based solutions, our approach has the chance to locate patches for more vulnerabilities since it considers broad correlation features (even weak ones) between a vulnerability and code commits. Besides, it uses a learning-based design to assign the weights to these correlation features, so it has a better tolerance for incomplete and incorrect vulnerability information. Further, even if our approach does not put the correct security patch at the first position of the ranked results, users can verify the ranked results one by one just as what they do with the search engine. Therefore, our approach can meet a better trade-off between the coverage of the security patches and the manual efforts involved.

3.2 Correlation Features

Our approach takes the vulnerability information as input and outputs the ranked code commits based on their correlations with the vulnerability. To acquire the vulnerability information, we refer to the CVE/NVD databases. Since the NVD covers more information than the CVE, we mainly choose the NVD page of a vulnerability as input to extract correlated vulnerability features. To formulate the correlation features between the code commits and a given vulnerability, we first investigate what types of information an NVD page contains and then check if these information may also be described by developers in the patch commits. As presented in Table 2, we conclude four groups of correlation features that may be commonly shared between the vulnerability information and the patch commits. Each feature group depicts the correlations between a commit and a vulnerability from one perspective. From the four feature groups, we further formulate *22* correlation features. We elaborate on these features below.

❶ *Vulnerability Identifier.* We consider two types of vulnerability identifiers — CVE-ID and software-specific Bug-ID. Mostly, disclosed vulnerabilities are publicly referred to by the CVE-ID. In addition, before being granted a CVE-ID, a vulnerability may be assigned with a software-specific Bug-ID which is used internally to track the life cycle of the vulnerability.

❷ *Vulnerability Location.* The location of a vulnerability is depicted using the *file* or the *function* that includes it. We recognize 6 features between the vulnerability and the code commit from this perspective. At the file level, we divide the files modified by a commit into two categories, namely, those that are mentioned in the vulnerability description and those that are unrelated to the description. We count the number of each category and calculate the ratio of the files that are shared between the vulnerability and the commit, which contributes to 3 features. The other 3 function-level features are calculated in a similar way.

❸ *Vulnerability Type.* It indicates the type/impact of the vulnerability, such as buffer-overflow, denial-of-service. We extract 2 features between a vulnerability and a code commit. First, we propose vulnerability type relevance that depicts the relevance of

Table 2: The features that are used by PatchScout to represent the correlations between a vulnerability and a code commit.

Feature Group	Features	Description
Vulnerability Identifier	CVE-ID	Whether the code commit mentions the CVE-ID of the target vulnerability.
	Software-specific Bug-ID	Whether the code commit mentions the software-specific Bug-ID in the NVD Page.
Vulnerability Location	Same Function Num	# of functions that appear in both code commit and NVD description.
	Same Function Ratio	# of same functions / # of functions modified by the code commit.
	Unrelated Function Num	# of functions that appear in code commit but not mentioned in the NVD description.
	Same File Num	# of files that appear in both code commit and NVD description.
	Same File Ratio	# of same files / # of files modified by the code commit.
	Unrelated File Num	# of files that appear in code commit but not mentioned in the NVD description.
Vulnerability Type	Vulnerability Type Relevance	The relevance of the vulnerability type-related texts between NVD information and commit message.
	Patch Likelihood	The probability of a commit to be a security patch.
Vulnerability Descriptive Texts	Shared-Vul-Msg-Word[1] Num	# of shared words between NVD description and commit message.
	Shared-Vul-Msg-Word Ratio	# of Shared-Vul-Msg-Words / # of words in NVD description.
	Max of Shared-Vul-Msg-Word Frequency	The max of the frequencies for all Shared-Vul-Msg-Words.
	Sum of Shared-Vul-Msg-Word Frequency	The sum of the frequencies for all Shared-Vul-Msg-Words.
	Average of Shared-Vul-Msg-Word Frequency	The average of the frequencies for all Shared-Vul-Msg-Words.
	Variance of Shared-Vul-Msg-Word Frequency	The variance of the frequencies for all Shared-Vul-Msg-Words.
	Shared-Vul-Code-Word[2] Num	# of shared words between NVD description and code diff.
	Shared-Vul-Code-Word Ratio	# of Shared-Vul-Code-Words / # of words in NVD description.
	Max of Shared-Vul-Code-Word Frequency	The max of the frequencies for all Shared-Vul-Code-Words.
	Sum of Shared-Vul-Code-Word Frequency	The sum of the frequencies for all Shared-Vul-Code-Words.
	Average of Shared-Vul-Code-Word Frequency	The average of the frequencies for all Shared-Vul-Code-Words.
	Variance of Shared-Vul-Code-Word Frequency	The variance of the frequencies for all Shared-Vul-Code-Words.

[1] Shared-Vul-Msg-Word: shared words between *NVD description* and *commit message*.
[2] Shared-Vul-Code-Word: shared words between *NVD description* and *code diff*.

the vulnerability type-related texts between the NVD vulnerability information and the commit message. Second, from the code diff aspect, we calculate patch likelihood to represent the probability of a commit to be a security patch.

❹ *Vulnerability Descriptive Texts.* It considers the vulnerability features that generally describe some types of vulnerability information such as critical variables, vulnerability trigger conditions, and vulnerability causes. We use the shared words between the vulnerability information and code commits to represent their correlations. As shown in Table 2, we calculate 6 statistical features from the shared words between vulnerability description and commit message, and the other 6 statistical features from the shared words between vulnerability description and code diff. As we will elaborate later (see §4), some meaningless words (e.g., stop words) are removed before calculation.

4 PATCHSCOUT DESIGN

This section presents PatchScout, which leverages the proposed *vulnerability-commit correlation ranking* to facilitate the locating of security patches for disclosed OSS vulnerabilities. We first introduce the workflow of PatchScout, and then elaborate its key modules.

Workflow. Given a target vulnerability, PatchScout takes the NVD database and the code repository as input and then ranks all the commits in the repository according to their correlations with the given vulnerability. There are mainly three phases:

(1) *Information Extraction.* PatchScout extracts some basic information elements from both NVD pages and code commits, which are used to generate features;
(2) *Feature Generation.* PatchScout generates the correlation features (that are introduced in §3.2) between a vulnerability and a code commit from the extracted information elements;
(3) *Commits Ranking.* PatchScout trains a RankNet-based model with the generated correlation features, to rank all the code commits based on their relevance to a specified vulnerability.

Table 3: The elements extracted from different sources.

Information Source	Extracted Element	Feature Goup[1] (Used By)	Extraction[2] Method
NVD Page	*description*	VDT	Extract
	vulnerability identifier	VID	Pattern
	file location	VL	Pattern
	function location	VL	NER
	vulnerability type	VT	NER, Extract
	vulnerability impact	VT	NER
Commit Message	*message*	VDT	Extract
	vulnerability identifier	VID	Pattern
	vulnerability type	VT	NER
	vulnerability impact	VT	NER
Commit Code	*code diff*	VDT, VT	Extract
	file location	VL	Pattern
	function location	VL	Pattern

[1] VDT represents *vulnerability descriptive texts*; VID represents *vulnerability identifier*; VL represents *vulnerability location*; VT represents *vulnerability type*.
[2] Extract: this information can be directly extracted; Pattern: extract information via pattern-matching; NER: extract information via named-entity recognition.

4.1 Information Extraction

As shown in Table 3, PatchScout extracts 8 kinds of information elements from NVD pages and code commits (including commit message and commit code), which are further used in the phase of feature generation. In particular, we adopt pattern-matching and named-entity recognition (NER) [28] to extract these information elements from these sources. ❶ From NVD pages, we directly extract the vulnerability description, CVE-ID, and vulnerability type (CWE). Further, from the extracted vulnerability description, we further extract file location via pattern-matching and leverage NER to identify function location, vulnerability type, and vulnerability impact. Besides, we also use pattern-matching to extract the software-specific bug-ID from the reference URLs of an NVD page as the complementary vulnerability identifier. ❷ From the commit message, we extract vulnerability identifier via pattern-matching

and identify vulnerability type and vulnerability impact via NER. ❸ From the commit code, we can directly extract the code diff and the file location and function location can be extracted via pattern-matching.

During the information extraction phase, we mainly use pattern-matching and NER, which are detailed in the below:

- *Pattern-matching*. We summarize common patterns of some information elements (file location, function location, software-specific bug-ID) and use regular expressions to extract such elements. For example, we match file suffix (e.g., .c, .h, .cpp, etc.) to identify file location in vulnerability description with the regular expression: ([a-zA-Z0-9]|-|_|/)+\.(cpp|cc|cxx|cp|CC|hpp|hh|C|c|h).
- *Named-entity Recognition.* In order to construct a training set, we collect 600 NVD descriptions and 164 patch commit messages and manually label the vulnerability type, vulnerability impact, and function location in these texts. Thereafter, we train an NER model on this dataset and apply the trained model in PatchScout to extract these elements.

4.2 Feature Generation

From the 8 kinds of information elements extracted between a vulnerability and a code commit, PatchScout further generates 22 correlation features (as listed in Table 2).

4.2.1 Vulnerability Identifier & Vulnerability Location. As described in Table 2, the features in the *vulnerability identifier* group and *vulnerability location* group are easy to generate. For vulnerability identifier group, we can directly determine whether a code commit and an NVD page refer to the same CVE-ID or the same software-specific bug-ID based on the extracted elements of vulnerability identifier. For vulnerability location group, based on the extracted elements of file/function location, we count the shared elements between a code commit and an NVD page and compute the 6 vulnerability location features as their definitions in Table 2.

4.2.2 Vulnerability Type Relevance. For a vulnerability, its type is usually mentioned in the vulnerability description and the CWE information of its NVD page, as well as in the commit message of its patch. Meanwhile, the impact of a vulnerability which is closely related to its vulnerability type, may also be mentioned in its NVD page and patch commit. Therefore, we use the two kinds (i.e., the vulnerability type and the vulnerability impact) of vulnerability type-related texts in an NVD page and a code commit to predict their vulnerability type relevance. These features help PatchScout to narrow down the search scope of patch commits.

Taxonomy of Type Relevance. As introduced in §4.1, PatchScout has extracted vulnerability type-related entities (i.e., vulnerability type and impact) from NVD pages and commit messages. However, little is known about what kind of relevance may exist between these entities. To this end, we conduct a study to find it out.

First, we randomly select 500 vulnerabilities that cover 47 CWEs in our training set (see §6.1) and collect a set of 1,219 vulnerability type-related entities from their NVD pages and security patch commits. Second, we normalize each entity to a word bag. Specifically, the normalization process consists of splitting the extracted entities into word sequences, removing the stop words, stemming, lemmatizing, and substituting synonym on the remaining words with the help of Natural Language Toolkit (NLTK) [47]. Third, we group the entities with the same word bag, which generates 31 entity groups. Note that there are 41 entities that do not belong to any group. Finally, from the 31 entity groups and 41 entities, we manually summarize three kinds of relevance that may exist between every two entity:

- *Inclusion relationship* describes the relationship between two vulnerability type entities or two vulnerability impact entities. There are two situations: 1) the vulnerability type/impact described by the two entities is the same or quite similar (e.g., buffer overflow and buffer overrun); 2) the vulnerability type/impact described by one entity is covered by the other entity (e.g., stack buffer overflow and buffer overflow).
- *Causality relationship* depicts the relationship between a vulnerability type entity and a vulnerability impact entity, when the former one may lead to the latter one. For example, there is a causality relation between stack-overflow and denial-of-service.
- *Irrelevance relationship* represents that there is no relation between the two entities.

Generating the Type Relevance Feature. Given the taxonomy of type relevance, PatchScout generates the vulnerability type relevance feature between a vulnerability and a code commit from two sets of vulnerability type-related entities (i.e., one is extracted from the NVD page and the other is extracted from the commit message) and represents this feature with a three-tuple which indicates the proportion of every kind of relevance between the two sets. There are three steps in generating this feature: 1) it transforms the two sets of entities into two sets of normalized word bags (as what we do in the above study); 2) it enumerates every word bag in the two sets to identify the relevance between a word bag in one set and another word bag in the other set (the identification method is explained later); 3) based on the frequencies of every kind of relevance, it computes the proportion for each of them and encodes these proportions into a three-tuple. To be specific, we identify the relevance between two entities (word bags) as follows:

(1) We use set operation to test if there exists an inclusion relation between two word bags.
(2) Since most (96.64%=1,178/1,219) of the collected vulnerability type-related entities in the previous study can be grouped into 31 unique normalized word bags, it is affordable to manually label the causality relationships between the 240 (16×15) word bag pairs. As shown in Table 13 (in §A.3), there are 16 vulnerability type groups (e.g., buffer overflow, integer overflow, use after free) and 15 vulnerability impact groups (e.g., denial of service, segmentation fault), and we label 150 causality relations between them. Based on these labels, it is straightforward to test if there exists a causality relation between two word bags. For those unlabelled word bags (3.36%), we simply consider there is no causality relation between them.
(3) If there is no causality or inclusion relation between two word bags, they are considered to be irrelevant.

4.2.3 Patch Likelihood. Code commits in an OSS project have various purposes, such as fixing performance bugs, fixing vulnerabilities, functionality updates, code clean-up [34]. As presented in Table 2, we use the patch likelihood feature to represent the

probability of a code commit to be a security patch, which helps to put the patch commits in front of other commits.

With a similar purpose, Wang *et al.* [68] have proposed a learning-based classification algorithm to identify security patches from OSS code commits. Their classification is based on the changed code lines and program elements (e.g., conditional statement, function call, etc) in a commit. Inspired by this work, PatchScout also leverages a learning-based approach to predict the patch likelihood of a code commit. Our work differs from this one in two aspects. First, from the perspective of the used features, we introduce two new features (detailed in §A.1) and collect a set of 62 features (see Table 9 in §A.1) from a code commit for patch likelihood prediction. Second, different from existing works [68] which aim to identify only security patch commits, we want to predict the patch likelihood for every commit.

In short, PatchScout predict the patch likelihood of a code commit in three steps. First, it generates the text features (No.1-8 in Table 9) from the texts in the code diff. Second, it performs an AST-based code diff analysis to identify the added/removed/updated/moved program elements and syntactic hunks, and then generates the syntactic features (No.9-62 in Table 9) from them. Finally, it trains 5 binary classification models [17, 18, 32, 35, 57] with these features and gives a patch likelihood for every code commit. To construct a training set, we use 943 security patch commits and 943 non-security patch commits that are verified in our preliminary study (see §2).

4.2.4 Vulnerability Descriptive Texts. The features in the group of vulnerability descriptive texts are used to capture the textual relevance between vulnerability descriptions and code commits via their shared words. Specifically, we collect shared words from two separated groups: one is between vulnerability description and commit message, and the other one is between vulnerability description and commit code. The shared words are identified in two steps. First, we split the descriptive texts with non-letter characters and then remove the stop words (e.g., prepositions and articles). Second, we take the intersection of two word sets to identify the shared words between them. Since a word may appear several times in a text, we not only count the shared words but also count the frequency of each shared word. Based on these words, we calculate 12 statistical features according to the definition in Table 2.

4.3 Commits Ranking

With the generated 22 correlation features, PatchScout trains a machine learning model to rank the code commits. To be specific, we choose to use RankNet [19], a pairwise learning-to-rank algorithm for code commits ranking based on two observations. First, we observe that code commit ranking is a classification problem on an extremely imbalanced dataset, where only few commits are security patches (aka positive cases) for a given vulnerability and the rest of the commits are all negative cases. Though classifying imbalance data is quite challenging [33, 48], RankNet is shown to be a promising solution to tackle the class imbalance problem [26]. Second, RankNet has been demonstrated its effectiveness in real-world ranking problems, such as Web page ranking [22], search engine personalization [62] and product recommendation [40].

Technically, RankNet trains a neural-network-based scoring model to give a score for every object (e.g., code commit), and then ranks all the objects based on their scores. In particular, to train a RankNet model, we need to prepare a set of object pairs $<x_i, x_j>$ with labels (i.e., whether x_i or x_j is a correct patch commit). Thereafter, RankNet initializes a neural network with random parameters and trains the model based on the labeled object pairs. In all, the training process consists of the following steps:

(1) For each fed object pair $<x_i, x_j>$, we calculate the *true probability* that x_i ranks higher than x_j as:

$$\overline{P}_{ij} = \begin{cases} 1 & if\ x_i\ is\ patch\ and\ x_j\ is\ not \\ 0.5 & both\ x_i\ and\ x_j\ are\ (non-)patches \\ 0 & if\ x_j\ is\ patch\ and\ x_i\ is\ not \end{cases}$$

(2) By representing an object as a feature vector, the neural network gives a score for every object. With the given scores for an object pair $<s_i, s_j>$ (s_i for x_i, s_j for x_j), we calculate the *learned probability* that x_i should be ranked higher than x_j as:

$$P_{ij} = \frac{1}{1 + e^{-(s_i - s_j)}}$$

(3) With the *true probability* ($\overline{P}_{ij}$) and the *learned probability* (P_{ij}), a cross entropy cost function C is calculated as:

$$C = -\overline{P}_{ij} log P_{ij} - (1 - \overline{P}_{ij}) log(1 - P_{ij})$$

(4) Using this cost function, the neural network scoring model is trained to minimize the cost in the training set.

5 IMPLEMENTATION

We implement a prototype of PatchScout, which contains 2,243 lines of Python code and 451 lines of Java code. Specifically, we leverage GitPython [4] to traverse code commits in OSS code repositories, and build an NER model based on NeuroNER [28] to extract vulnerability type-related entities and function entities from NVD description and commit messages. For type relevance analysis, we use NLTK (a Python NLP toolkit) [47] to normalize entities. We use its *nltk.tokenize* module to split the entities into word sequences, filter the stop words according to the stop word list in the *nltk.corpus* module, use its *nltk.stem* module to perform stemming and lemmatizing, and leverage the WordNet database [66] in the *nltk.corpus* module to find the synonyms and perform synonym substitution. For patch likelihood prediction, we use Gumtree [30] to perform AST-based code diff analysis. For commit ranking, we implement the RankNet algorithm [19] on PyTorch [55]. Our prototype is extensible to support vulnerabilities in various programming languages, since most of the features we select are language-independent. The major extension effort is to train a new NER-based parser and enhance the AST-based code diff analysis for the new language.

6 EVALUATION

This section evaluates PatchScout in locating patches for disclosed OSS vulnerabilities. It first introduces the experimental setup and then presents the evaluation results on the effectiveness of PatchScout, the contributions of the proposed features, and the possibility of enhancing PatchScout by predicting the bug fix files.

6.1 Experimental Setup

Our evaluation requires a number of OSS vulnerabilities including their security patches for the purpose of training and testing. In our preliminary study (see §2), we have collected a large number of OSS CVEs as a vulnerability dataset. Therefore, our evaluation also uses this dataset.

Model Training. PatchScout needs labelled samples to train the commit ranking model. Since we have manually verified 2,200 potential patch commits (see experiment-2 of §2) in our preliminary study, we use the verified patch commits as positive samples. Specifically, after removing the duplicate patch commits, the positive samples consist of 943 unique patch commits as well as 943 disclosed vulnerabilities. For each positive sample (x_{patch}), we randomly select 5,000 other commits from the Git repository as negative samples, and then construct 5,000 sample pairs <x_{patch}, x_{other_i}>, where x_{other_i} is a negative sample for x_{patch}. Note that when a repository contains less than 5,000 commits, we take all the other commits of the repository as negative samples in that case. In all, our training set has 3,329,286 sample pairs. We use an Ubuntu 16.04 64-bit machine (with 314 GB memory, 4 Intel Xeon Gold 5215 processors, and 1 GeForce RTX 2080Ti GPU) for model training. By feeding the training set into a RankNet algorithm, it takes about 3 hours to train the ranking model for PatchScout.

Testing Set. We construct a testing set consisting of 685 disclosed vulnerabilities and their patch commits within two steps. First, we randomly pick out 800 CVEs which do not overlap the training set. Second, we try our best to locate their patches, by not only taking the three intuitive approaches mentioned in §2 but also checking other resources, such as bug tracking reports and vulnerability-related code commits. At last, we successfully locate the security patches for 685 CVEs (belonging to 187 OSS), and use them as the testing set. The remaining 115 CVEs are considered as unpatched and thus ignored in the testing set. In all, three security researchers participate in constructing the testing set and double-checking all the security patches, which costs 240 man-hours.

Baselines. To the best of our knowledge, our work is the first one that provides a systematic way to locate the security patches of disclosed OSS vulnerabilities. As described in §2, existing works mainly adopt three intuitive methods for such a task. Therefore, we include these intuitive methods as the baselines. To further demonstrate the benefits of the ranking-based design, we also devise an enhanced keyword matching-based method as the baseline, named *M4*. It works as follows. First, it uses the same method as PatchScout to extract vulnerability identifiers, locations, and types from NVD pages. Second, to make a fair comparison, it uses the techniques in §4.2.2 to extend synonyms for extracted vulnerability types. Third, it uses the extended vulnerability types, identifiers, and locations as keywords to search the commits and ranks the matched commits by the number of keywords they hit. Note that for commits with the same number of matched keywords, they are ranked randomly and the experiment is repeated six times to reduce the effect of randomness.

Metrics. We use the following two metrics to evaluate the effectiveness of PatchScout and baselines.

- *Recall.* For M1, M2 and M3, if a patch is covered by their return results, we consider they successfully locate the patch. For PatchScout and M4, if a patch is in its top N (a parameter set by users) results, we consider the patch is successfully located and we name it as top-N recall.
- *Manual Efforts.* For M1, M2 and M3, since they are matching-based rather than ranking-based, all the matched commits have the same priority to the users. Since all the matched commits need to be manually verified, we use the number of matched commits to measure the involved manual efforts. When using PatchScout and M4, users check the ranked code commits one by one to look for a patch commit. Therefore, we use the number of commits that need to be manually checked before finding the patch commit to calculate the required manual efforts. In particular, if the correct security patch for a CVE is ranked R-th by PatchScout and the users are asked to check the top N commits given by PatchScout to locate the patch, the manual efforts for locating the security patch of this CVE is $min(R, N)$. Accordingly, the average involved manual efforts for n CVEs can be calculated as $\frac{\sum_{i=1}^{n} min(R_i,N)}{n}$.

Table 4: Performance of PatchScout and baselines on locating security patches.

Approaches	Recall	Manual Efforts
M1: searching with CVE-ID	11.53%	2.30
M2: checking commit-like URLs	40.00%	1.14
M3: checking patch-tagged URLs	31.53%	1.61
M1+M2+M3	47.59%	1.83
M4: enhanced keyword matching (N=1)	40.88%	1.00
M4: enhanced keyword matching (N=5)	61.80%	2.92
M4: enhanced keyword matching (N=30)	80.10%	9.37
PatchScout (N=1)	69.49%	1.00
PatchScout (N=5)	85.40%	1.86
PatchScout (N=30)	92.70%	4.32

6.2 Effectiveness

Table 4 presents the overall results. We find that PatchScout ranks the security patches of 69.49% CVEs at the first position. Moreover, by checking 4.32 commits on average over the top 30 commits ranked by PatchScout, we can locate the security patches for 92.70% CVEs. Since locating the security patches by searching numerous code commits is extremely time-consuming and laborious, the involved manual effort here is acceptably low. In contrast, even when combining all the three baselines (M1+M2+M3), only 47.59% patches can be located with 1.83 commits on average to be checked. Interestingly, if users are asked to only check the top 5 commits given by PatchScout, they only need to check 1.86 commits per CVE (close to that of M1+M2+M3), while they can locate 85.40% patches (37.81% more than M1+M2+M3). Though M4 is enhanced with the keywords extracted by PatchScout, it can only rank 40.88% patches at the first position. Even when we check the top 30 commits give by M4, we can only locate 80.10% patches but pay more than twice manual efforts than PatchScout. These results clearly demonstrate the strength of PatchScout on locating security patches for more vulnerabilities with less manual efforts.

False Negatives Breakdown. Though PatchScout has covered the patches for most of the disclosed vulnerabilities, it still fails on 7.30% (50) CVEs in the testing set, even when the parameter N is set to 30. We manually check all the 50 false negatives (FN) and discover the following two causes. In §8, we further discuss how to mitigate these FNs.

- *Low-quality vulnerability information in the NVD.* Although PatchScout utilizes broad correlations between the vulnerability information and the code commits, it requires to extract meaningful information. However, in some CVEs, we found their NVD pages only contain low-quality information that describes no specific features of a vulnerability. In these cases, PatchScout cannot extract useful features (those depicted in Table 2) from the NVD, thus failing to rank their security patch commits in front of other commits.
- *Giant commits.* We find that developers sometimes merge multiple code updates for different purposes into a single (giant) commit. If a giant commit contains a security patch, its vulnerability-irrelevant information and code behaviors weaken its relevance to the corresponding security vulnerability. It makes PatchScout lower its ranking and eventually causes FNs.

Patch Distribution at Each Rank. To measure the distribution of patches at each rank, we also vary the parameter N to test the performance of PatchScout. The results are presented in Table 10 (in §A.3). We find that the more efforts users put in checking the ranked commits, the more security patches they can locate. Overall, 77.66% patch commits can be located by checking at most the top 2 commits ranked by PatchScout, and more than 90% patch commits can be located by checking about 3 commits on average (3.04 commits per CVE when N = 15). These results clearly show that PatchScout effectively balances the coverage of the security patches and the involved manual efforts, by ranking the code commits that are the most likely security patches to the front.

6.3 Feature Group Contributions

As presented in Table 2, PatchScout uses four correlative feature groups to rank code commits. To measure the contribution of each feature group to the overall performance, we train four new ranking models that drop one feature group each. The four weakened models are trained and tested with the same dataset of PatchScout.

As Table 5 shows, the *vulnerability identifier* contributes the most. For example, it increases the recall by 24.09% when N = 1. This is because the vulnerability identifier is specific to a vulnerability and may reveal the most direct relation between a vulnerability and its patch commit. Similarly, the feature group of *vulnerability location* also contributes significantly to the overall performance, since it helps to exclude a large number of irrelevant code commits. Besides, we find that even weak correlations (e.g., vulnerability type, vulnerability descriptive texts) between a vulnerability and a code commit play very important roles in locating the patch commits. They not only help to locate the security patches for more vulnerabilities, but also help to rank them to more front positions. In summary, we find each feature group effectively improves the performance of PatchScout in helping to locate the patch commits.

6.4 Possibility of Leveraging Bug Fix Prediction

Security patch is also a type of bug fix. Existing works on predicting the fix of a bug [36, 41, 43, 45, 79] might help to further reduce the search scope of security patches. We conduct several experiments to explore this possibility. Our experiments focus on two questions: ❶ How can an ideal bug fix predictor help PatchScout? ❷ How do the existing bug fix prediction techniques help?

Table 5: Contribution of each feature group.

Top N	Drop Identifier	Drop Location	Drop Type	Drop Texts
1	45.40% (24.09% ↓)	58.54% (10.95% ↓)	62.48% (7.01% ↓)	61.90% (7.59% ↓)
2	57.96% (19.70% ↓)	66.13% (11.53% ↓)	74.31% (3.35% ↓)	73.14% (4.52% ↓)
3	63.94% (18.54% ↓)	69.49% (12.99% ↓)	78.54% (3.94% ↓)	77.23% (5.25% ↓)
4	67.45% (16.64% ↓)	71.82% (12.27% ↓)	80.00% (4.09% ↓)	80.29% (3.80% ↓)
5	70.07% (15.33% ↓)	72.99% (12.41% ↓)	81.90% (3.50% ↓)	82.34% (3.06% ↓)
6	71.97% (14.45% ↓)	74.45% (11.97% ↓)	82.48% (3.94% ↓)	84.23% (2.19% ↓)
7	73.28% (14.02% ↓)	75.77% (11.53% ↓)	83.07% (4.23% ↓)	84.96% (2.34% ↓)
8	73.87% (14.01% ↓)	76.64% (11.24% ↓)	84.23% (3.65% ↓)	85.55% (2.33% ↓)
9	75.18% (13.29% ↓)	77.37% (11.10% ↓)	85.26% (3.21% ↓)	85.99% (2.48% ↓)
10	75.91% (12.85% ↓)	77.96% (10.80% ↓)	85.26% (3.50% ↓)	86.42% (2.34% ↓)

The percentage in parentheses represents the reduction in recall for this model compared to the original model that takes all the feature groups.

Leveraging an ideal bug fix predictor. We first investigate whether an ideal bug fix predictor can help PatchScout reduce the search scope. In particular, we use the patched file(s) in a security patch as an ideal bug fix predictor. For the 209 CVEs whose patches are not ranked as top-1 by PatchScout (§6.2), we analyze the non-patch commits that are ranked at a more front position than the patch commits. Under a conservative strategy, PatchScout can filter out 49.80% non-patch commits that do not modify any file predicted by the ideal bug fix predictor. Besides, if PatchScout only keeps the commits that modify the same files predicted by the ideal bug fix predictor, 79.57% non-patch commits can be removed. By leveraging the predicted patched files as the *vulnerability location* information during the training and testing of PatchScout, its top-1 and top-5 recall increases by 8.03% and 5.84% respectively. In summary, we find an ideal bug fix predictor can largely reduce the search scope of security patches.

Leveraging existing works in bug fix prediction. We then leverage two existing bug fix prediction techniques, IR-based bug localization [36, 41, 45, 79] and usual suspect [41, 52], to improve PatchScout. First, the IR-based bug localization studies how to predict the files to be fixed from a bug report. In particular, we use `BugLocator`[3] [79] (an information retrieval-based bug localization tool) to analyze the bug report of a vulnerability, and we use the predicted to-be-fixed files as a supplement to the *vulnerability location* information extracted from the NVD page. In our testing set, `BugLocator` successfully predicts a patch file in its top-5 ranked results for 347 CVEs (50.66%), and for 409 CVEs (59.71%) in its top-10 ranked results. However, we find the top-1 recall of PatchScout drops 2.92% when using the top-5 prediction results of `BugLocator` and drops more (4.23%) when using the top-10 prediction results of `BugLocator`. The main reason is that `BugLocator` cannot accurately predict the patch files for a vulnerability. The wrongly predicted patch files decrease the correlation similarity with the true patch commits, but they may increase the correlation similarity with the non-patch commits. Note the recent efforts [36, 45] apply deep learning to improve the performance of bug localization, successfully predicting a patch file in its top-10 ranked results for more than 80% bugs. Since these tools are not available, we cannot directly evaluate them. Meanwhile, we believe PatchScout can hardly benefit from them either, since their accuracy is still not high.

Second, according to the observation of [41, 52], most bug fixes are applied on a small fraction of components, which means the

[3]The source code of BugLocator is available at https://github.com/exatoa/Bench4BL. Besides, we enhance it to support C/C++ projects by introducing a new code parser.

files that have been fixed before may have a higher chance to be fixed further. To verify whether such usual suspect can help filter non-patch commits, we perform an experiment. First, we select 3 OSS projects (*Linux Kernel*, *Wireshark*, and *tcpdump*) that have more than 100 CVEs in the OSS vulnerability dataset collected in §2 and manually locate the patch files for these CVEs. Second, we collect the top-10 most frequently fixed files for each project and use these files as a supplement to the *vulnerability location* information extracted from the NVD page. Then, we use PatchScout to locate the security patches for the 112 CVEs of the 3 OSS projects in our testing set. The results show that the top-1 recall of PatchScout drops 10.71%. Again, we find the reason is that the usual suspect of vulnerable components is not accurate enough to help locate the patch for a specific vulnerability.

Takeaway. Based on the above experiments, we find that an ideal bug fix predictor greatly helps PatchScout and the accuracy of bug fix prediction techniques significantly affect their effectiveness in improving PatchScout. These findings motivate more accurate bug fix prediction techniques.

7 PATCH DEPLOYMENT ACROSS BRANCHES

We now illustrate the security benefits of PatchScout via conducting a study on the patch deployment practice across branches.

7.1 Study Design

The development of OSS is usually managed within branches and each branch corresponds to a specific version. When a vulnerability is reported, a security patch is developed on its master branch. Since some old versions may be also affected and still-in-use, developers should deploy the security patch to these branches/versions too, though introducing some maintenance overhead at the same time. However, to the best of our knowledge, little is known about such patch deployment practice across branches in the real world.

By ranking a set of code commits according to their relevance to a vulnerability, PatchScout also helps to locate the security patch of a given vulnerability under a specific branch. Therefore, we apply PatchScout to perform the first study on patch deployment practice across different branches. Specifically, our study is organized from three aspects: *patch deployment status*, *patch backporting*, and *patch deployment lag*. As we will show later, this study is hard to be conducted without the help of PatchScout.

CVEs and Branches. To study patch deployment practice on multi-branches, we mainly consider popular OSS projects. In particular, we select four popular C/C++ OSS (Linux Kernel [6], Wireshark [11], QEMU [7], and FFmpeg [3]) and one Java OSS (Jenkins [5]) as our study targets. To construct a set of CVE-branch pairs, for each OSS, we randomly select 45 CVEs that were reported after 2016 and collect branches in their Git repositories that have new code commits after 2016. In all, our study consists of 3,735 CVE-branch pairs (225 CVEs and 83 branches). Note that we only consider stable/release branches in the study. The detailed information about these branches and CVEs is listed in Table 14 (in §A.3).

Patch Collection. For each CVE-branch pair, PatchScout is applied to help locate the patch commit for the specific CVE and branch. We also extend PatchScout to support Java OSS vulnerabilities according to the instructions in §5. Based on the ranked code commits by PatchScout, we manually check the top 30 candidates to locate the security patch. In all, we successfully locate the security patches for 1,985 CVE-branch pairs with 33 man-hours. The results are shown in Table 6.

We further investigate each CVE-branch pair that has no security patch located by PatchScout. There are mainly four situations: 1) the branch is not affected by the vulnerability[4] (184 *not-affected* cases); 2) the branch has already been out-of-maintenance[5] before the vulnerability is disclosed (1,206 *not-maintained* cases); 3) the branch is affected and maintained but has not applied the patch (150 *not-patched* cases); 4) the branch is patched but PatchScout fails to locate their patch commits (210 cases). For these 210 cases, we manually locate the patch commits from the Git repository with 21 man-hours. At last, we collect the security patches for 2,195 CVE-branch pairs. Note PatchScout helps locate 90.43% patch commits, and only 49.52% (1,087) patch commits can be located by combining the three baseline methods introduced in §6. Also, 1,985 patches are located with the help of PatchScout in 33 hours (1 minute per patch), and the locating of the other 210 patches costs 21 hours (6 minutes per patch) when PatchScout cannot help. It demonstrates the importance of PatchScout in facilitating such studies.

7.2 Patch Deployment Status

Not-patched CVEs and Branches. As mentioned above, there are 150 CVE-branch pairs that are maintained but not-patched. Such situations are very dangerous because these branches are still under maintenance (which means they may still have users in-use) but forget to apply some security patches. We further breakdown these not-patched pairs in Table 7. Surprisingly, we find such not-patched situations are quite popular from the perspective of both CVE entries and branches. For example, there are 90 (40.00%) CVE entries having at least one not-patched branch and 43 (51.81%) branches having at least one not-patched CVE. This finding shows that a great many branches are ignored during the patch deployment process, which brings great risks to the end-users.

It is well-believed that vulnerabilities with higher risks will be more seriously treated by OSS developers/maintainers. To verify this assumption, we explore the correlation between the not-patched ratio of each CVE with its CVSS score (as presented in Figure 3 in §A.3). We are surprised to find that 30 (33.33%) not-patched CVEs belong to high-risk vulnerabilities (CVSS score >= 7.0), and the distribution of not-patched ratio is independent of the CVSS score of each CVE. The not-patched ratio of the severest CVE is not significantly lower than those of other vulnerabilities. It implies that the maintainers may not take the vulnerability severity into consideration when propagating patches across branches.

Affected Versions in CVE/NVD. Mu *et al.* [50] and Dong *et al.* [29] find that the information about the affected versions in CVE/NVD may be incomplete. Since the patch presence information of a branch (version) indicates if it is affected, we can use this information to check the correctness of the affected versions of a vulnerability in CVE/NVD. To be more specific, if the security patch is found at a branch (version) and the branch exists before the vulnerability report date, this branch is thought to be affected. We

[4]We manually confirm the branch is not affected by the vulnerability.
[5]We find the branch has no code commits after the vulnerability disclosed date.

Table 6: Patch deployment status on different branches.

Software	# of CVEs	# of Branches	# of CVE-Branches	# of patched CVE-Branches	# (%) of patches identified by PatchScout	# (%) of patches identified with M1+M2+M3
Linux Kernel	45	30	1,350	709	671 (94.64%)	439 (61.92%)
Wireshark	45	8	360	232	214 (92.24%)	59 (25.43%)
QEMU	45	14	630	353	342 (96.88%)	264 (74.79%)
FFmpeg	45	12	540	388	362 (93.30%)	206 (53.09%)
Jenkins	45	19	855	513	396 (77.19%)	119 (23.20%)
Total	225	83	3,735	2,195	1,985 (90.43%)	1,087 (49.52%)

Table 7: Not-patched CVEs and branches for each OSS.

Software	# of not-patched CVE-branches	# (%) of CVEs with not-patched branches	# (%) of branches with not-patched CVEs
Linux Kernel	38 (2.81%)	20 (44.44%)	18 (60.00%)
Wireshark	16 (4.44%)	10 (22.22%)	8 (100%)
QEMU	33 (5.24%)	28 (62.22%)	7 (50.00%)
FFmpeg	50 (9.26%)	19 (42.22%)	7 (58.33%)
Jenkins	13 (1.52%)	13 (28.89%)	3 (15.79%)
Total	150 (4.02%)	90 (40.00%)	43 (51.81%)

use the 2,195 CVE-branch pairs with identified patches to manually check the affected versions in CVE/NVD.

In all, we find 152 affected versions for 38 CVEs are missed (detailed in Table 11). We report all these missed affected versions to the CVE community, which has confirmed all our findings and has updated the descriptions about the affected versions for these 38 CVEs. Though PatchScout is not designed for such a task (i.e., finding miss-reported affected versions), this experiment renders its benefits to the community from another angle.

7.3 Patch Backporting

Security patches are usually developed on a certain branch and then deployed to other affected branches. During the cross-branch patch deployment, developers may need to adjust the original patch according to the code of the target branch. We investigate all the patches that are located in §7.1 to measure the efforts in cross-branch patch deployment. Specifically, from the 2,195 CVE-branches with identified patches, we find 725 unique patch commit IDs. On average, 3.22 unique patches are developed to fix a vulnerability on different branches.

Efforts in Patch Backporting. For each CVE, we recognize its first-developed patch as the original patch and treat others as backported ones. Based on the difference between the original patch and a backported patch, we classify three types of efforts that are required in backporting patches. The first type is to directly apply the original patch without any change. The second type needs to adjust the line number of the original patch according to the target branch, but the code diff is not changed. The third type needs code adaption and customization, since the code diff of the backported patch is different from the original one.

As shown in Table 8, developers can directly apply the patch for 32.20% branches and need to adjust the code line number for 61.00% cases. For the remaining 6.80% cases, developers have to customize the patch code to fit the target branch.

Difficulties in Code Adaption. To understand the difficulties in adapting the patch code, we analyze all the 34 patches that need code adaption in Table 8.

Table 8: Percentage of different backported patches.

Software	Type-1[1]	Type-2[2]	Type-3[3]
Linux Kernel	60 (28.57%)	136 (64.76%)	14 (6.67%)
Wireshark	26 (28.89%)	53 (58.89%)	11 (12.22%)
QEMU	12 (41.38%)	16 (55.17%)	1 (3.45%)
FFmpeg	63 (36.84%)	100 (58.48%)	8 (4.68%)
Jenkins[4]	N/A	N/A	N/A
Total	161 (32.20%)	305 (61.00%)	34 (6.80%)

1. Direct deployment. 2. Line NO. adjustment. 3. Code adaption.
4. Jenkins has no backported patches among different branches.

- *Updating code irrelevant elements (11 cases).* The code difference between an original patch and the backported one is caused by code irrelevant elements, such as comments and indentation.
- *Merging multiple commits (3 cases).* When developers merge the original security patch and other code updates into a single commit and deploy it on a branch, the resulting backported commit is different from the original security patch in code behaviors. Though the merged commit has different code from the original patch commit, the major technical difficulties of this type of code adaption lie in merging several code commits.
- *Fitting different code context (20 cases).* We also find that the context of the pre-patched code may differ on different branches. As a result, developers have to understand the vulnerability logic and put more efforts into adapting the original patch to a new code context on the target branch.

In conclusion, our study makes the first attempt to shed some light on the efforts and difficulties in patch backporting with real-world OSS projects and CVEs. Our study classifies different types of patch backporting situations and recognizes different levels of code adaption, which could facilitate some follow-up research, such as assisting patch backporting and identifying backported patches.

7.4 Patch Deployment Lag

In addition to the difference among patch commits on different branches, their patch deployment time also varies significantly.

Vulnerability Disclosure Time vs. First Patch Time. First, we collect the time when the first patch was applied and compare it with the vulnerability disclosure time (as presented in Figure 4 in §A.3) . Specifically, the average time lag is -24.21 days, which means developers usually respond to the reported vulnerabilities in time. Furthermore, there are 65 CVEs whose first patch is deployed after the vulnerability disclosure and 24 CVEs are fixed even after one month. These delays in vulnerability fixing lengthen the attack window of those vulnerabilities.

First Patch Time vs. Last Patch Time. We also collect the propagation time of a patch from the earliest branch to the last

branch and present the results in Table 12 (in §A.3). On average, it takes 68.20 days to propagate the first patch to other branches. Besides, the median of the propagation time is 25 days and the longest propagation time is 702 days. The patch delay across branches prolongs the risks of these vulnerabilities over end-users.

7.5 Takeaway

Our study demonstrates that a large fraction of affected branches are still unpatched and other branches, while patched, suffer from a quite long patch lag. We propose some suggestions to improve the patch deployment process across branches.

Verifying affected versions of a vulnerability. Our study finds that CVE/NVD misses many affected versions, which may ultimately mislead the developers to forget deploying patches on those branches. Correct information about the affected versions would help developers to locate the candidate branches for patch deployment. Therefore, how to verify the affected versions for a vulnerability becomes an important problem. To this end, code clone detection [37, 42, 46] and directed fuzzing [16, 23] may be used here to locate the potential affected branches.

Managing patch deployment progress. We suppose that software developers/maintainers might intentionally prioritize the patch deployment process due to constrained resources. However, we find no obvious clue for such assumption in our study. This means the current patch deployment practice among multiple branches is lack of management. Therefore, it calls for automatic tools to check the patch deployment status across branches, so the developers pay more attention to deploy patches to all affected branches in time.

Easing patch backporting. As discussed in §7.3, the original patch sometimes requires some extra efforts (either line number adjustment or code adaption) to deploy on other branches, which increases the cost of patch propagating. It indicates that some automatic techniques are needed to ease the process of patch backporting, e.g., adjusting the line number when deploying patches, testing the applicability of a security patch to a branch.

8 DISCUSSION

Identifying generic patches and then linking back to specific vulnerabilities. SPIDER [49] and [68] aim to identify generic patches. However, we may face coverage issues when directly adopting them, since the recall of [68] is 79.6% and SPIDER only identifies 55.37% CVE patches as safe-patches. Instead, PatchScout enhances [68] to measure the patch likelihood of each commit and uses it as a feature to improve patch locating and ranking.

Collecting vulnerability information from more sources. As shown in our evaluation, sometimes the quality of the vulnerability information in NVD is low, which limits the effectiveness of PatchScout in locating security patches. We report our initial results in extracting more information from bug reports in §A.2. In the future, we plan to collect information from more vulnerability databases, such as SecurityTracker [9], SecurityFocus [8].

Deeply analyzing the commit code. The code in a patch commit contains much useful information to understand the vulnerability. However, we only use the AST of the code diff to predict its patch likelihood. In fact, more vulnerability-related information can be extracted from the code commit by deeply analyzing its code. For example, we can leverage static analysis [64] or symbolic execution [21, 58] techniques to analyze whether the commit introduces a boundary check on an array, which is highly relevant to fixing a buffer overflow vulnerability.

Locating patch commits for other kinds of bugs. In addition to locating security patches, our general idea of ranking code commits can be also applied to locating the patch commits for other bug types (e.g., performance bugs, functional bugs). For example, by analyzing some performance bugs reported by Jin *et al.* [39], we also find broad correlations (e.g., bug identifier, bug location, descriptive texts) between the performance bugs and their patches. In the future, we will explore the possibility of *bug-commit correlation ranking* to locate patch commits for other kinds of bugs.

Locating vulnerability-introducing commits. There may be two kinds of commits related to a vulnerability in the code repository: a *vulnerability-introducing* commit which introduces a vulnerability and a patch commit which fixes a vulnerability. Intuitively, both kinds of commits may be located by PatchScout. In fact, we do not find a vulnerability-introducing commit during our evaluation and study. This is because different from the patch commits, the correlations between the vulnerability-introducing commits and the vulnerabilities are indirect and implicit.

9 RELATED WORK

Bug Fix Prediction. Predicating the fix of a bug is a popular research topic in the field of mining software repositories (MSR). Anvik *et al.* [14] propose a learning-based approach to predict the developers that should fix a bug. Information-retrieval-based bug localization techniques [12, 36, 41, 43, 45, 60, 67, 69, 74, 76, 79] suggest the code components (e.g., files, functions) that are likely to be fixed for a bug by mining bug reports and source code. While these works intend to ease the patch development by predicting some properties of a patch, PatchScout focuses on easing the locating of patch commits in the code repository. Furthermore, as demonstrated in our evaluation, locating security patches requires more accurate correlating than predicting bug fix. To provide effective patch locating, PatchScout considers broad correlation features and incorporates a learning-based rank system.

Security Patch Identification. Existing works also make some attempts to collect security patches. Xu *et al.* [73] propose a pattern matching-based approach to identify security patches in binaries. Tian *et al.* [65] and Wang *et al.* [68] leverage machine learning to identify security patches at the source code level. Specifically, Tian *et al.* [65] extract features from both commit messages and code diffs, while Wang *et al.* [68] focus on mining more code diff features for patch identification. Further, SPIDER [49] introduces the concept of *safe patch* for security patch identification. All these works identify the security patches but cannot associate them with the vulnerabilities they fix. Different from these works, PatchScout supports locating the security patches of a specified vulnerability.

Security Patch Study. Since security patches are widely used, they have become an important target to study. Rescorla *et al.* [59] analyze OpenSSL security patches to understand users' responses to vulnerabilities. Yin *et al.* [75] perform a study on incorrect patches to categorize incorrect patch patterns and understand the causes behind them. Zhong *et al.* [78] and Soto *et al.* [63] study a

large-scale of patches to guide automatic bug repair. Further, Li *et al.* [44] conduct a comprehensive study on the development life cycle of security patches. Dai *et al.* [27] perform a study on patch deployment practice on downstream binaries with the support of a patch presence testing tool. However, as far as we know, there is no study about the practice of patch deployment across different branches. With the help of PatchScout, this paper could perform the first study on patch deployment practice across branches.

10 CONCLUSION

This paper presents PatchScout, a software tool to help locate the security patches for disclosed OSS vulnerabilities in their code repositories. The key idea of PatchScout is to transform the search problem of locating security patches into a ranking problem on code commits. To rank patch commits in front of other commits, PatchScout proposes a *vulnerability-commit correlation ranking* mechanism, which exploits the broad correlations between a vulnerability and a code commit. The ranking-based design enables PatchScout to locate more security patches and meet a balance between the patch coverage and the manual efforts involved. Our evaluation on 685 OSS vulnerabilities shows that PatchScout significantly outperforms all existing methods in both patch coverage and manual workload. With the help of PatchScout, this paper performs the first study on patch deployment practice across branches with 5 popular OSS projects and 225 CVEs, drawing many interesting findings and new research directions.

Availability. We plan to release the source code of PatchScout, and the dataset in our evaluation and the patch deployment study.

REFERENCES

[1] 2019. What are the most secure programming languages? https://www.whitesourcesoftware.com/most-secure-programming-languages/.
[2] 2020. Open source vulnerability management report. https://www.whitesourcesoftware.com/open-source-vulnerability-management-report/.
[3] 2021. FFmpeg. https://git.ffmpeg.org/ffmpeg.
[4] 2021. GitPython. https://github.com/gitpython-developers/GitPython.
[5] 2021. Jenkins. https://github.com/jenkinsci/jenkins.
[6] 2021. Linux Kernel. https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git.
[7] 2021. QEMU. https://git.qemu.org/git/qemu.git.
[8] 2021. Security Focus. https://www.securityfocus.com/.
[9] 2021. Security Tracker. https://securitytracker.com/.
[10] 2021. Vulnerable code database Project. https://github.com/google/vulncode-db.
[11] 2021. Wireshark. https://gitlab.com/wireshark/wireshark.
[12] Shayan A. Akbar and Avinash C. Kak. 2019. SCOR: Source Code Retrieval with Semantics and Order. In *Proceedings of the 16th International Conference on Mining Software Repositories (MSR)* (Montreal, Quebec, Canada). 1–12.
[13] Gautam Altekar, Ilya Bagrak, Paul Burstein, and Andrew Schultz. 2005. OPUS: Online Patches and Updates for Security. In *Proceedings of the 14th USENIX Security Symposium (USENIX Security)*. 287–302.
[14] John Anvik, Lyndon Hiew, and Gail C. Murphy. 2006. Who Should Fix This Bug?. In *Proceedings of the 28th International Conference on Software Engineering (ICSE)* (Shanghai, China). 361–370.
[15] Jeff Arnold and M. Frans Kaashoek. 2009. Ksplice: Automatic Rebootless Kernel Updates. In *Proceedings of the 4th ACM European Conference on Computer Systems (EuroSys)* (Nuremberg, Germany). New York, NY, USA, 187–198.
[16] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. 2017. Directed Greybox Fuzzing. In *Proceedings of the 24th ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 2329–2344.
[17] Léon Bottou. 2010. Large-scale Machine Learning with Stochastic Gradient Descent. In *Proceedings of the 19th International Conference on Computational Statistics (COMPSTAT)*. 177–186.
[18] Leo Breiman. 1996. Bagging predictors. *Machine learning* (1996), 123–140.
[19] Chris Burges, Tal Shaked, Erin Renshaw, Ari Lazier, Matt Deeds, Nicole Hamilton, and Greg Hullender. 2005. Learning to Rank Using Gradient Descent. In *Proceedings of the 22nd International Conference on Machine Learning (ICML)* (Bonn, Germany). 89–96.
[20] Christopher JC Burges. 2010. From RankNet to LambdaRank to LambdaMart: An Overview. *Learning* (2010).
[21] Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI)*. 209–224.
[22] Olivier Chapelle and Yi Chang. 2011. Yahoo! Learning to Rank Challenge Overview. In *Proceedings of the Learning to Rank Challenge*. 1–24.
[23] Hongxu Chen, Yinxing Xue, Yuekang Li, Bihuan Chen, Xiaofei Xie, Xiuheng Wu, and Yang Liu. 2018. Hawkeye: Towards a Desired Directed Grey-Box Fuzzer. In *Proceedings of the 25th ACM SIGSAC Conference on Computer and Communications Security (CCS)*.
[24] Yue Chen, Yulong Zhang, Zhi Wang, Liangzhao Xia, Chenfu Bao, and Tao Wei. 2017. Adaptive Android Kernel Live Patching. In *Proceedings of the 26th USENIX Security Symposium (USENIX Security)*.
[25] MITRE Corporation. 2021. Common Vulnerabilities and Exposures. https://cve.mitre.org/.
[26] Ricardo Cruz, Kelwin Fernandes, Jaime S Cardoso, and Joaquim F Pinto Costa. 2016. Tackling Class Imbalance with Ranking. In *Proceedings of the 2016 International Joint Conference on Neural Networks (IJCNN)*. 2182–2187.
[27] Jiarun Dai, Yuan Zhang, Zheyue Jiang, Yingtian Zhou, Junyan Chen, Xinyu Xing, Xiaohan Zhang, Xin Tan, Min Yang, and Zhemin Yang. 2020. BScout: Direct Whole Patch Presence Test for Java Executables. In *Proceedings of the 29th USENIX Security Symposium (USENIX Security)*. 1147–1164.
[28] Franck Dernoncourt, Ji Young Lee, and Peter Szolovits. 2017. NeuroNER: an Easy-to-use Program for Named-entity Recognition based on Neural Networks. In *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing: System Demonstrations (EMNLP)*. 97–102.
[29] Ying Dong, Wenbo Guo, Yueqi Chen, Xinyu Xing, Yuqing Zhang, and Gang Wang. 2019. Towards the Detection of Inconsistencies in Public Security Vulnerability Reports. In *Proceedings of the 28th USENIX Security Symposium (USENIX Security)*.
[30] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. 2014. Fine-grained and Accurate Source Code Differencing. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering (ASE)*. 313–324.
[31] Qian Feng, Rundong Zhou, Yanhui Zhao, Jia Ma, Yifei Wang, Na Yu, Xudong Jin, Jian Wang, Ahmed Azab, and Peng Ning. 2019. Learning Binary Representation for Automatic Patch Detection. In *Proceedings of the 16th IEEE Annual Consumer Communications & Networking Conference (CCNC)*. 1–6.
[32] Nir Friedman, Dan Geiger, and Moises Goldszmidt. 1997. Bayesian Network Classifiers. *Machine learning* (1997), 131–163.
[33] Guo Haixiang, Li Yijing, Jennifer Shang, Gu Mingyun, Huang Yuanyue, and Gong Bing. 2017. Learning from Class-imbalanced Data: Review of Methods and Applications. *Expert Systems with Applications* (2017), 220–239.
[34] Abram Hindle, Daniel M. German, Michael W. Godfrey, and Richard C. Holt. 2009. Automatic Classification of Large Changes into Maintenance categories. In *Proceedings of the 17th International Conference on Program Comprehension (ICPC)*. 30–39.
[35] Tin Kam Ho. 1995. Random Decision Forests. In *Proceedings of the 3rd International Conference on Document Analysis and Recognition (ICDAR)*. 278–282.
[36] Xuan Huo, Ming Li, and Zhi-Hua Zhou. 2016. Learning Unified Features from Natural and Programming Languages for Locating Buggy Source Code. In *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence (IJCAI)* (New York, New York, USA). 1606–1612.
[37] Jiyong Jang, Abeer Agrawal, and David Brumley. 2012. ReDeBug: Finding Unpatched Code Clones in Entire OS Distributions. In *Proceedings of the 33rd IEEE Symposium on Security and Privacy (S&P)*. 48–62.
[38] Zheyue Jiang, Yuan Zhang, Jun Xu, Qi Wen, Zhenghe Wang, Xiaohan Zhang, Xinyu Xing, Min Yang, and Zhemin Yang. 2020. PDiff: Semantic-based Patch Presence Testing for Downstream Kernels. In *Proceedings of the 27th ACM SIGSAC Conference on Computer and Communications Security (CCS)*.
[39] Guoliang Jin, Linhai Song, Xiaoming Shi, Joel Scherpelz, and Shan Lu. 2012. Understanding and Detecting Real-World Performance Bugs. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (Beijing, China). 77–88.
[40] Shubhra Kanti Karmaker Santu, Parikshit Sondhi, and ChengXiang Zhai. 2017. On Application of Learning to Rank for E-Commerce Search. In *Proceedings of the 40th International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR)* (Shinjuku, Tokyo, Japan). 475–484.
[41] Dongsun Kim, Yida Tao, Sunghun Kim, and Andreas Zeller. 2013. Where should we fix this bug? a two-phase recommendation model. *IEEE transactions on software Engineering* 39, 11 (2013), 1597–1610.
[42] Seulbae Kim, Seunghoon Woo, Heejo Lee, and Hakjoo Oh. 2017. VUDDY: A Scalable Approach for Vulnerable Code Clone Discovery. In *Proceedings of the 38th IEEE Symposium on Security and Privacy (S&P)*. 595–614.

[43] An Ngoc Lam, Anh Tuan Nguyen, Hoan Anh Nguyen, and Tien N. Nguyen. 2017. Bug Localization with Combination of Deep Learning and Information Retrieval. In *Proceedings of the 25th International Conference on Program Comprehension (ICPC)* (Buenos Aires, Argentina). 218–229.
[44] Frank Li and Vern Paxson. 2017. A Large-scale Empirical Study of Security Patches. In *Proceedings of the 24th ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 2201–2215.
[45] Hongliang Liang, Lu Sun, Meilin Wang, and Yuxing Yang. 2019. Deep learning with customized abstract syntax tree for bug localization. *IEEE Access* 7 (2019), 116309–116320.
[46] Zhen Liu, Qiang Wei, and Yan Cao. 2017. Vfdetect: A Vulnerable Code Clone Detection System Based on Vulnerability Fingerprint. In *Proceedings of the 3rd Information Technology and Mechatronics Engineering Conference (ITOEC)*. 548–553.
[47] Edward Loper and Steven Bird. 2002. NLTK: the Natural Language Toolkit. In *Proceedings of the ACL-02 Workshop on Effective tools and methodologies for teaching natural language processing and computational linguistics-Volume 1*. 63–70.
[48] Victoria López, Alberto Fernández, Salvador García, Vasile Palade, and Francisco Herrera. 2013. An Insight into Classification with Imbalanced Data: Empirical Results and Current Trends on Using Data Intrinsic Characteristics. *Information sciences* (2013), 113–141.
[49] Aravind Machiry, Nilo Redini, Eric Camellini, Christopher Kruegel, and Giovanni Vigna. 2020. SPIDER: Enabling Fast Patch Propagation In Related Software Repositories. In *Proceedings of the 41th IEEE Symposium on Security and Privacy (S&P)*. 1562–1579.
[50] Dongliang Mu, Alejandro Cuevas, Limin Yang, Hang Hu, Xinyu Xing, Bing Mao, and Gang Wang. 2018. Understanding the Reproducibility of Crowd-reported Security Vulnerabilities. In *Proceedings of the 27th USENIX Security Symposium (USENIX Security)*.
[51] Collin Mulliner, Jon Oberheide, William Robertson, and Engin Kirda. 2013. Patchdroid: Scalable Third-party Security Patches for Android Devices. In *Proceedings of the 29th Annual Computer Security Applications Conference (ACSAC)*. 259–268.
[52] Stephan Neuhaus, Thomas Zimmermann, Christian Holler, and Andreas Zeller. 2007. Predicting Vulnerable Software Components. In *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS)* (Alexandria, Virginia, USA). 529–540.
[53] U.S. National Institute of Standards and Technology. 2016. NVD - CVE-2016-4417. https://nvd.nist.gov/vuln/detail/CVE-2016-4417.
[54] U.S. National Institute of Standards and Technology. 2021. National Vulnerability Database. https://nvd.nist.gov/home.cfm.
[55] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Proceedings of the 32nd Advances in Neural Information Processing Systems (NIPS)*. 8026–8037.
[56] Henning Perl, Sergej Dechand, Matthew Smith, Daniel Arp, Fabian Yamaguchi, Konrad Rieck, Sascha Fahl, and Yasemin Acar. 2015. 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 (CCS)*. 426–437.
[57] John C Platt. 1999. Advances in Kernel Methods. Chapter Fast Training of Support Vector Machines using Sequential Minimal Optimization. *MIT Press, Cambridge, MA, USA* (1999), 185–208.
[58] Sebastian Poeplau and Aurélien Francillon. 2020. Symbolic Execution with SymCC: Don't Interpret, Compile!. In *Proceedings of the 29th USENIX Security Symposium (USENIX Security)*. 181–198.
[59] Eric Rescorla. 2003. Security holes... Who cares?. In *Proceedings of the 12th USENIX Security Symposium (USENIX Security)*. 75–90.
[60] Ripon K. Saha, Matthew Lease, Sarfraz Khurshid, and Dewayne E. Perry. 2013. Improving Bug Localization Using Structured Information Retrieval. In *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)* (Silicon Valley, CA, USA). 345–355.
[61] Armin Sarabi, Ziyun Zhu, Chaowei Xiao, Mingyan Liu, and Tudor Dumitraş. 2017. Patch Me If You Can: A Study on the Effects of Individual User Behavior on the End-Host Vulnerability State. In *Proceedings of the 2017 International Conference on Passive and Active Network Measurement (PAM)*. 113–125.
[62] Yang Song, Hongning Wang, and Xiaodong He. 2014. Adapting Deep RankNet for Personalized Search. In *Proceedings of the 7th ACM International Conference on Web Search and Data Mining (WSDM)* (New York, New York, USA). 83–92.
[63] Mauricio Soto, Ferdian Thung, Chu-Pan Wong, Claire Le Goues, and David Lo. 2016. A Deeper Look into Bug Fixes: Patterns, Replacements, Deletions, and Additions. In *Proceedings of the 13th Working Conference on Mining Software Repositories (MSR)*.
[64] Yulei Sui and Jingling Xue. 2016. SVF: Interprocedural Static Value-Flow Analysis in LLVM. In *Proceedings of the 25th International Conference on Compiler Construction (CC)*. 265–266.
[65] Yuan Tian, Julia Lawall, and David Lo. 2012. Identifying Linux Bug Fixing Patches. In *Proceedings of the 34th International Conference on Software Engineering (ICSE)*. 386–396.
[66] Princeton University. 2010. WordNet. https://wordnet.princeton.edu/.
[67] Shaowei Wang and David Lo. 2014. Version History, Similar Report, and Structure: Putting Them Together for Improved Bug Localization. In *Proceedings of the 22nd International Conference on Program Comprehension (ICPC)* (Hyderabad, India). 53–63.
[68] Xinda Wang, Kun Sun, Archer Batcheller, and Sushil Jajodia. 2019. Detecting "0-Day" Vulnerability: An Empirical Study of Secret Security Patch in OSS. In *Proceedings of the 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 485–492.
[69] Ming Wen, Rongxin Wu, and Shing-Chi Cheung. 2016. Locus: Locating Bugs from Software Changes. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)* (Singapore, Singapore). 262–273.
[70] Wireshark. 2016. Patch of CVE-2016-4417. https://gitlab.com/wireshark/wireshark/-/commit/c31425f9ae15067e26ccc6183c206c34713cb256.
[71] Yang Xiao, Bihuan Chen, Chendong Yu, Zhengzi Xu, Zimu Yuan, Feng Li, Binghong Liu, Yang Liu, Wei Huo, Wei Zou, and Wenchang Shi. 2020. MVP: Detecting Vulnerabilities using Patch-Enhanced Vulnerability Signatures. In *Proceedings of the 29th USENIX Security Symposium (USENIX Security)*. 1165–1182.
[72] Yifei Xu, Zhengzi Xu, Bihuan Chen, Fu Song, Yang Liu, and Ting Liu. 2020. Patch Based Vulnerability Matching for Binary Programs. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. 376–387.
[73] Zhengzi Xu, Bihuan Chen, Mahinthan Chandramohan, Yang Liu, and Fu Song. 2017. SPAIN: security patch analysis for binaries towards understanding the pain and pills. In *Proceedings of the 39th International Conference on Software Engineering (ICSE)*. 462–472.
[74] Xin Ye, Razvan Bunescu, and Chang Liu. 2014. Learning to Rank Relevant Files for Bug Reports Using Domain Knowledge. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)* (Hong Kong, China). 689–699.
[75] Zuoning Yin, Ding Yuan, Yuanyuan Zhou, Shankar Pasupathy, and Lakshmi Bairavasundaram. 2011. How Do Fixes Become Bugs?. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering (ESEC/FSE)*. 26–36.
[76] Klaus Changsun Youm, June Ahn, Jeongho Kim, and Eunseok Lee. 2015. Bug localization based on code change histories and bug reports. In *Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 190–197.
[77] Hang Zhang and Zhiyun Qian. 2018. Precise and Accurate Patch Presence Test for Binaries. In *Proceedings of the 27th USENIX Security Symposium (USENIX Security)* (Baltimore, MD, USA). USA, 887–902.
[78] Hao Zhong and Zhendong Su. 2015. An Empirical Study on Real Bug Fixes. In *Proceedings of the 37th IEEE International Conference on Software Engineering (ICSE)*. 913–923.
[79] Jian Zhou, Hongyu Zhang, and David Lo. 2012. Where Should the Bugs Be Fixed? - More Accurate Information Retrieval-Based Bug Localization Based on Bug Reports. In *Proceedings of the 34th International Conference on Software Engineering (ICSE)* (Zurich, Switzerland). 14–24.

A APPENDIX

A.1 Features in Predicting Patch Likelihood

To predict the patch likelihood of a code commit, PatchScout leverages a learning-based approach. It collects a set of 62 features (see Table 9) from a code commit. Compared to [68], we introduce the following two new features.

- *Update and movement of program elements.* Wang *et al.* recognize the code diff as a sequence of additions and deletions of program elements. However, we observe that some additions and deletions should be recognized as updates and movements to illustrate the real purposes of these code changes. Figure 2 gives two examples: Figure 2 (a) fixes an infinite recursion vulnerability by updating the branch condition (line 16 to line 18); while Figure 2 (b) fixes a heap buffer overflow by moving the condition check (line 9, 10) to the inside of the loop (line 16, 17). In these

Table 9: Features to predict the patch likelihood of a commit.

No.	Feature
1	# of changed files
2	# of changed functions
3	# of hunks
4	# of same hunk
5 - 8	# of added/removed/total/net lines
9 - 12	# of added/removed/total/net conditional statements
13 - 16	# of added/removed/total/net loops
17 - 20	# of added/removed/total/net logical expressions
21 - 24	# of added/removed/total/net functions
25 - 28	# of added/removed/total/net function calls
29 - 32	# of added/removed/total/net assignments
33 - 36	# of added/removed/total/net memory related operations
37 - 40	# of added/removed/total/net exits
41 - 44	# of added/removed/total/net returns
45	# of updated conditional statements
46	# of updated loops
47	# of updated logical expressions
48	# of updated function calls
49	# of updated memory related operations
50	# of updated returns
51	# of updated operands in conditional statement
52	# of updated operands in loop
53	# of updated operands in logical expression
54	# of updated operands in function call
55	# of updated operands in memory related operation
56	# of updated operands in return
57	# of moved conditional statements
58	# of moved loops
59 - 62	# of added/removed/updated/moved syntactic hunks

two cases, simply recognizing the code changes as addition and deletion would overlook the real semantic (update and movement) behind it. Therefore, we introduce the features of updates and movements on program elements (No.45-58 in Table 9) into the classification model. These features are collected by matching the patterns of additions and deletions.

- *Syntactic hunks.* The discreteness of the code diff may help to differ patch commits from other commits. To represent the discreteness of the code diff, Wang *et al.* use textual-level hunks (i.e., a chunk of code consisting of continuous modified code lines and several unmodified code lines around them). However, textual-level changes in code lines do not directly reflect the syntactic-level changes in program elements, so we also consider how discrete the code diff is at the syntactic level. In particular, we introduce 4 syntactic hunk features (No.59-62 in Table 9) which use continuous added/removed/updated/moved program elements to represent the discreteness at the syntactic level. We extract these features by simply counting the syntactic hunks.

```
diff --git a/src/frompnm.c b/src/frompnm.c
index 86d0c03..de73766 100644
--- a/src/frompnm.c
+++ b/src/frompnm.c
@@ -36,13 +36,15 @@ pnm_get_line(unsigned char *p, unsigned char *end, ...
    int n;

    do {
+       /* read the line */
        for (n = 0 ; p < end && *p >= ' '; p++) {
            if (n < 255) {
                line[n++] = *p;
            }
        }

-       if (p < end && *p == '\n') {
+       /* skip invald characters */
+       if (p < end && *p < ' ') {
            p++;
        }

```

(a) Patch Commit of CVE-2019-11024

```
diff --git a/coders/sgi.c b/coders/sgi.c
index 236bf4cb9..415598122 100644
--- a/coders/sgi.c
+++ b/coders/sgi.c
@@ -953,8 +953,6 @@ static MagickBooleanType WriteSGIImage(const ImageInfo ...
  assert(image->signature == MagickCoreSignature);
  if (image->debug != MagickFalse)
    (void) LogMagickEvent(TraceEvent,GetMagickModule(),"%s",image->filename);
- if ((image->columns > 65535UL) || (image->rows > 65535UL))
-   ThrowWriterException(ImageError,"WidthOrHeightExceedsLimit");
  assert(exception != (ExceptionInfo *) NULL);
  ...
  do
  {
    ...
+   if ((image->columns > 65535UL) || (image->rows > 65535UL))
+     ThrowWriterException(ImageError,"WidthOrHeightExceedsLimit");
    ...
```

(b) Patch Commit of CVE-2019-19948

Figure 2: Examples of security patch fixing vulnerability by update or movement.

A.2 Enhance PatchScout with Bug Reports

We conduct an experiment to measure the possibility of using the vulnerability information in bug reports to enhance PatchScout. Specifically, from the 1,628 (=943+685) CVEs in our training set and testing set, we find 1,391 bug reports in their NVD pages. From these reports, we successfully extract more *vulnerability identifier*, *vulnerability location*, *vulnerability type* information for 641 CVEs, using the same method in §4.1. We then enhance PatchScout to use these new features during training and testing. However, we find that the effectiveness of PatchScout decreases a little after adopting the features extracted from bug reports. To be specific, the top-1 recall of PatchScout drops 0.88% and its top-10 recall drops 1.02%. We further investigate these new features and find that though more information is extracted from the bug reports, much of it is incorrect, e.g., bug reports usually contain stack traces which have many patch-irrelevant functions and files. In particular, we find that for 400 CVEs, the relevance between the true patch commit and the vulnerability information degrades in the corresponding feature dimensions. It turns out that the form of information in the bug reports is more complex, and a more accurate information extractor is required to enhance PatchScout.

A.3 Supplementary Tables and Figures

Table 10: Performance of PatchScout with different N.

Top N	Recall	Manual Efforts	Top N	Recall	Manual Efforts
1	69.49%	1.00	8	87.88%	2.27
2	77.66%	1.31	9	88.47%	2.39
3	82.48%	1.53	10	88.76%	2.51
4	84.09%	1.70	15	90.36%	3.04
5	85.40%	1.86	20	91.24%	3.50
6	86.42%	2.01	25	91.82%	3.93
7	87.30%	2.14	30	92.70%	4.32

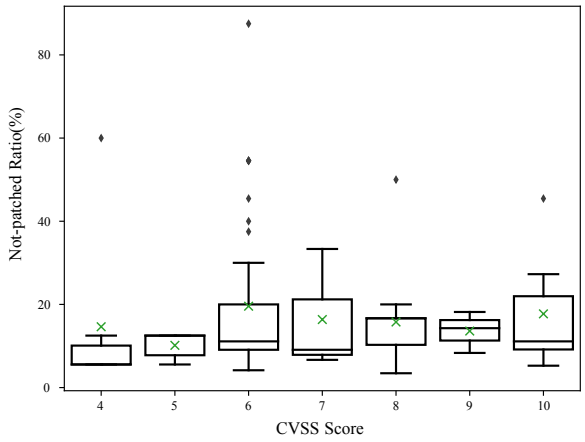

Figure 3: The not-patched ratios among different CVEs.

Table 11: Miss-reported affected versions in CVE/NVD.

Software	# of CVE-branches with affected versions	# of CVEs with affected versions
Linux Kernel	73/1,350 (5.41%)	15/45 (33.33%)
Wireshark	0/360 (0.00%)	0/45 (0.00%)
QEMU	3/630 (0.48%)	3/45 (6.67%)
FFmpeg	76/540 (14.07%)	20/45 (44.44%)
Jenkins	0/855 (0.00%)	0/45 (0.00%)
Total	152/3,735 (4.07%)	38/225 (16.89%)

Table 12: Propagation time (day) between the first patch and the last patch on all affected branches.

Software	Minimum	Median	Average	Maximum
Linux Kernel	1	52	67.33	249
Wireshark	0	0	23.70	702
QEMU	0	106.5	116.38	228
FFmpeg	0	38	84.44	420
Jenkins[1]	N/A	N/A	N/A	N/A
Total	0	25	68.20	702

[1] In Jenkins, there's only one unique patch on all patched branches for each vulnerability, which means there's no patch propagation.

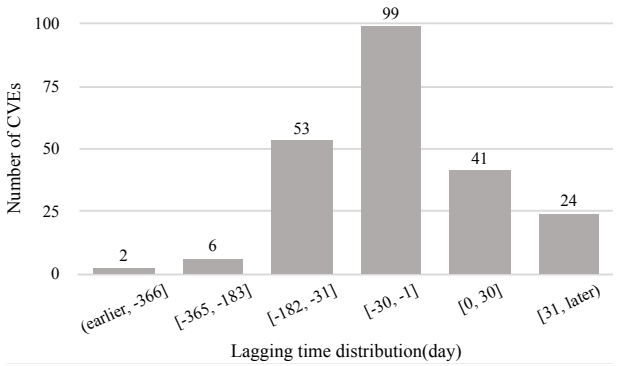

Figure 4: Lag between the time when the CVE is disclosed and the time of the first patch.

Table 13: Vulnerability type groups, vulnerability impact groups and the causality relations between them.

Vulnerability Type Groups (16)	Vulnerability Impact Groups (15)
overflow	denial of service, crash, segmentation fault, segmentation violation, memory issues, memory corruption, invalid memory access, code execution, unspecified impact, assertion failure
buffer overflow	denial of service, crash, segmentation fault, segmentation violation, memory issues, memory corruption, invalid memory access, code execution, unspecified impact, assertion failure
integer overflow	denial of service, crash, unspecified impact, assertion failure
heap overflow	denial of service, crash, segmentation fault, segmentation violation, memory issues, memory corruption, invalid memory access, code execution, unspecified impact, assertion failure
stack overflow	denial of service, crash, segmentation fault, segmentation violation, memory issues, memory corruption, invalid memory access, code execution, unspecified impact, assertion failure
off by one	denial of service, crash, segmentation fault, segmentation violation, memory issues, memory corruption, invalid memory access, code execution, unspecified impact, assertion failure
use after free	denial of service, crash, segmentation fault, segmentation violation, memory issues, memory corruption, invalid memory access, code execution, obtain sensitive information, unspecified impact, assertion failure
double free	denial of service, crash, segmentation fault, segmentation violation, memory issues, memory corruption, invalid memory access, code execution, obtain sensitive information, unspecified impact, assertion failure
infinite loop	denial of service, memory issues, memory leak, memory corruption, memory consumption, unspecified impact, stack consumption
out of bound access	denial of service, crash, segmentation fault, segmentation violation, memory issues, memory corruption, invalid memory access, code execution, obtain sensitive information, unspecified impact, assertion failure, bus error
out of bound write	denial of service, crash, segmentation fault, segmentation violation, memory issues, memory corruption, invalid memory access, code execution, unspecified impact, assertion failure, bus error
null pointer dereference	denial of service, crash, segmentation fault, segmentation violation, memory issues, memory corruption, invalid memory access, unspecified impact, assertion failure
out of bound read	memory issues, memory corruption, invalid memory access, obtain sensitive information, unspecified impact, assertion failure, bus error
miss bound check	denial of service, crash, segmentation fault, segmentation violation, memory issues, memory corruption, invalid memory access, code execution, obtain sensitive information, unspecified impact, assertion failure
divide by zero	denial of service, crash, segmentation fault, segmentation violation, unspecified impact, assertion failure
race condition	denial of service, crash, segmentation fault, segmentation violation, memory issues, memory corruption, invalid memory access, code execution, obtain sensitive information, unspecified impact, assertion failure

Table 14: Detailed Dataset Information of CVEs and Branches for Patch Deployment Study across Branches (see §7).

Software	CVEs					Branches
Linux Kernel	CVE-2017-5577	CVE-2017-6214	CVE-2017-7273	CVE-2017-7889	CVE-2017-8063	linux-3.2.y linux-3.10.y linux-3.12.y linux-3.16.y linux-3.18.y linux-4.1.y linux-4.4.y linux-4.8.y linux-4.9.y linux-4.10.y linux-4.11.y linux-4.12.y linux-4.13.y linux-4.14.y linux-4.15.y linux-4.16.y linux-4.17.y linux-4.18.y linux-4.19.y linux-4.20.y linux-5.0.y linux-5.1.y linux-5.2.y linux-5.3.y linux-5.4.y linux-5.5.y linux-5.6.y linux-5.7.y linux-5.8.y master
	CVE-2017-10911	CVE-2017-11473	CVE-2017-12193	CVE-2017-16530	CVE-2017-16995	
	CVE-2018-5332	CVE-2018-5803	CVE-2018-7492	CVE-2017-18204	CVE-2017-18216	
	CVE-2018-10940	CVE-2018-13099	CVE-2018-13096	CVE-2018-13405	CVE-2018-16276	
	CVE-2018-19854	CVE-2019-11833	CVE-2019-12817	CVE-2018-20961	CVE-2019-15222	
	CVE-2019-15919	CVE-2019-15923	CVE-2019-15927	CVE-2019-17052	CVE-2019-19079	
	CVE-2019-19078	CVE-2019-19065	CVE-2019-19061	CVE-2019-19052	CVE-2019-19534	
	CVE-2019-19535	CVE-2019-19807	CVE-2020-9383	CVE-2020-11494	CVE-2020-11884	
	CVE-2020-12768	CVE-2020-13143	CVE-2020-13974	CVE-2020-14416	CVE-2020-15393	
Wireshark	CVE-2016-5359	CVE-2016-6507	CVE-2016-6509	CVE-2016-6512	CVE-2017-5597	master-1.12 master-2.0 master-2.2 master-2.4 master-2.6 master-3.0 master-3.2 master
	CVE-2017-6474	CVE-2017-7701	CVE-2017-7703	CVE-2017-7748	CVE-2017-7747	
	CVE-2018-7420	CVE-2018-7336	CVE-2018-7321	CVE-2018-9268	CVE-2018-9265	
	CVE-2018-9273	CVE-2018-9257	CVE-2018-9258	CVE-2018-9271	CVE-2018-11356	
	CVE-2018-11354	CVE-2018-14370	CVE-2018-14341	CVE-2018-14343	CVE-2018-16058	
	CVE-2018-18225	CVE-2018-19626	CVE-2019-5721	CVE-2019-5717	CVE-2019-5716	
	CVE-2019-9209	CVE-2019-9214	CVE-2019-10902	CVE-2019-10894	CVE-2019-10896	
	CVE-2019-10900	CVE-2019-10903	CVE-2019-12295	CVE-2019-13619	CVE-2019-19553	
	CVE-2020-7044	CVE-2020-7045	CVE-2020-9428	CVE-2020-9431	CVE-2020-13164	
QEMU	CVE-2016-4037	CVE-2016-6490	CVE-2016-6835	CVE-2016-6836	CVE-2016-7116	stable-2.5 stable-2.6 stable-2.7 stable-2.8 stable-2.9 stable-2.10 stable-2.11 stable-2.12 stable-3.0 stable-3.1 stable-4.0 stable-4.1 stable-4.2 master
	CVE-2016-7466	CVE-2016-7421	CVE-2016-9102	CVE-2016-9105	CVE-2016-9106	
	CVE-2017-5525	CVE-2017-5552	CVE-2017-5578	CVE-2017-5579	CVE-2017-5667	
	CVE-2017-5857	CVE-2017-5898	CVE-2017-5931	CVE-2017-5973	CVE-2017-5987	
	CVE-2017-6058	CVE-2017-7377	CVE-2017-8086	CVE-2017-8284	CVE-2017-18030	
	CVE-2018-15746	CVE-2018-17958	CVE-2018-18849	CVE-2018-19489	CVE-2018-20126	
	CVE-2018-20125	CVE-2018-20123	CVE-2018-20216	CVE-2019-5008	CVE-2019-3812	
	CVE-2019-6501	CVE-2019-6778	CVE-2018-20815	CVE-2019-12155	CVE-2019-13164	
	CVE-2019-15034	CVE-2019-20382	CVE-2020-11102	CVE-2020-11869	CVE-2020-13765	
FFmpeg	CVE-2016-6164	CVE-2016-6920	CVE-2016-10190	CVE-2016-10192	CVE-2017-7865	release/2.4 release/2.8 release/3.0 release/3.1 release/3.2 release/3.3 release/3.4 release/4.0 release/4.1 release/4.2 release/4.3 master
	CVE-2017-7862	CVE-2017-9990	CVE-2017-9992	CVE-2017-9994	CVE-2017-9991	
	CVE-2017-11399	CVE-2017-11719	CVE-2017-14058	CVE-2017-14170	CVE-2017-14169	
	CVE-2017-14171	CVE-2017-14767	CVE-2017-15672	CVE-2017-16840	CVE-2017-17081	
	CVE-2018-6621	CVE-2018-6912	CVE-2018-7557	CVE-2018-7751	CVE-2018-12459	
	CVE-2018-12458	CVE-2018-12460	CVE-2018-13301	CVE-2018-13300	CVE-2018-13303	
	CVE-2018-13302	CVE-2018-13305	CVE-2018-13304	CVE-2018-14394	CVE-2018-14395	
	CVE-2018-15822	CVE-2019-1000016	CVE-2019-9721	CVE-2019-9718	CVE-2019-11338	
	CVE-2019-12730	CVE-2019-17539	CVE-2019-17542	CVE-2020-12284	CVE-2020-13904	
Jenkins	CVE-2016-0788	CVE-2016-0789	CVE-2016-3725	CVE-2016-9299	CVE-2017-2600	stable-2.107 stable-2.121 stable-2.138 stable-2.150 stable-2.164 stable-2.176 stable-2.190 stable-2.19 stable-2.204 stable-2.222 stable-2.235 stable-2.249 stable-2.32 stable-2.46 stable-2.60 stable-2.7 stable-2.73 stable-2.89 master
	CVE-2017-2606	CVE-2017-2608	CVE-2017-2610	CVE-2017-2601	CVE-2017-2611	
	CVE-2017-2602	CVE-2017-1000362	CVE-2017-1000399	CVE-2017-1000393	CVE-2017-1000391	
	CVE-2017-1000355	CVE-2018-1000169	CVE-2018-1000193	CVE-2018-1000194	CVE-2018-1999003	
	CVE-2018-1999001	CVE-2018-1999044	CVE-2018-1000861	CVE-2018-1000862	CVE-2018-1000864	
	CVE-2018-1000408	CVE-2018-1000406	CVE-2018-1000409	CVE-2018-1000410	CVE-2018-1000407	
	CVE-2019-10406	CVE-2019-10405	CVE-2019-10404	CVE-2019-10401	CVE-2019-10384	
	CVE-2019-10383	CVE-2019-10353	CVE-2019-1003050	CVE-2019-1003049	CVE-2020-2161	
	CVE-2020-2105	CVE-2020-2104	CVE-2020-2103	CVE-2020-2102	CVE-2020-2162	