Tracking Patches for Open Source Software Vulnerabilities

Congying Xu*
School of Computer Science
Fudan University
Shanghai, China

Bihuan Chen*†
School of Computer Science
Fudan University
Shanghai, China

Chenhao Lu*
School of Computer Science
Fudan University
Shanghai, China

Kaifeng Huang*
School of Computer Science
Fudan University
Shanghai, China

Xin Peng*
School of Computer Science
Fudan University
Shanghai, China

Yang Liu
School of Computer Science and Engineering
Nanyang Technological University
Singapore

ABSTRACT

Open source software (OSS) vulnerabilities threaten the security of software systems that use OSS. Vulnerability databases provide valuable information (e.g., vulnerable version and patch) to mitigate OSS vulnerabilities. There arises a growing concern about the information quality of vulnerability databases. However, it is unclear what the quality of patches in existing vulnerability databases is; and existing manual or heuristic-based approaches for patch tracking are either too expensive or too specific to apply to all OSS vulnerabilities.

To address these problems, we first conduct an empirical study to understand the quality and characteristics of patches for OSS vulnerabilities in two industrial vulnerability databases. Inspired by our study, we then propose the first automated approach, Tracer, to track patches for OSS vulnerabilities from multiple knowledge sources. Our evaluation has demonstrated that i) Tracer can track patches for up to 273.8% more vulnerabilities than heuristic-based approaches while achieving a higher F1-score by up to 116.8%; and ii) Tracer can complement industrial vulnerability databases. Our evaluation has also indicated the generality and practical usefulness of Tracer.

CCS CONCEPTS

• **Information systems** → **Open source software**; • **Security and privacy** → **Vulnerability management**.

KEYWORDS

open source software, vulnerability patches, patch tracking

ACM Reference Format:
Congying Xu, Bihuan Chen, Chenhao Lu, Kaifeng Huang, Xin Peng, and Yang Liu. 2022. Tracking Patches for Open Source Software Vulnerabilities. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '22), November 14–18, 2022, Singapore, Singapore.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3540250.3549125

*Also with Shanghai Key Laboratory of Data Science, and Shanghai Collaborative Innovation Center of Intelligent Visual Computing.
†Bihuan Chen is the corresponding author.

ESEC/FSE '22, November 14–18, 2022, Singapore, Singapore

ACM ISBN 978-1-4503-9413-0/22/11…$15.00
https://doi.org/10.1145/3540250.3549125

1 INTRODUCTION

Open source software (OSS) provides the foundation for open source and proprietary applications. It allows developers to reuse functionalities instead of reinventing the wheel. As revealed by a recent report [3], 98% of applications contain OSS. However, security risks are also introduced with OSS. 84% of applications contain at least an OSS vulnerability in 2020, a 9% increase from the 75% in 2019; and each application contains an average of 158 OSS vulnerabilities [3]. Even worse, OSS vulnerabilities are detected at an increasing speed, nearly doubling in the last two years [54]. Hence, OSS vulnerability management has become more and more urgent.

Great efforts have been made to mitigate security risks in OSS vulnerabilities. Vulnerability databases play a significant role in these efforts by providing valuable information (e.g., description, vulnerable versions, and patches) for different vulnerability analysis tasks. CVE List [8] and NVD [41] are well-known public vulnerability databases, which even go beyond OSS vulnerabilities. They provide data feed of the entire database, and often serve as the main vulnerability source of industrial vulnerability databases, e.g., Black Duck [15], WhiteSource [62], Veracode [60] and Snyk [55]. These industrial vulnerability databases are specifically focused on OSS vulnerabilities.

Problem. While vulnerability databases are accumulating a massive collection of vulnerabilities, there arises an increasing concern about information quality of vulnerability databases. Nguyen and Massacci [39] and Dong et al. [13] revealed the unreliability of vulnerable version data in vulnerability databases. Chaparro et al. [4] and Mu et al. [36] showed the prevalence of missing reproducing steps in vulnerability descriptions. The missing or inaccurate information of vulnerability entries in vulnerability databases makes it challenging to timely mitigate OSS vulnerabilities in applications.

Patch is a valuable piece of information to capture a vulnerability. It enables not only OSS vulnerability mitigation (e.g., software composition analysis [44, 45, 61]), but also other security tasks (e.g., hot patch generation and deployment [14, 37, 66], patch presence testing [9, 26, 69] and vulnerability detection [7, 24, 31, 33, 34, 63]). On one hand, the accuracy of these tasks is affected if vulnerability patches are missing or inaccurate. However, the problem is that *it is unclear what the quality of patches in vulnerability databases is.*

On the other hand, apart from leveraging patches in vulnerability databases, these tasks track vulnerability patches by manual efforts [6, 9, 26, 44, 45, 51, 63, 66, 71], by heuristic rules like looking for commits in CVE references [14, 33, 34] and searching for CVE identifiers in commits [61, 67], or from security advisories that list the vulnerabilities and their patches for specific projects [24, 31, 37]. However, the problem is that *these patch tracking approaches are either too expensive or too specific to apply to all OSS vulnerabilities.*

Empirical Study. To address the first problem, we conduct an empirical study to understand the quality and characteristics of patches for OSS vulnerabilities in two industrial vulnerability databases (i.e., Veracode [60] and Snyk [55]). On the basis of a dataset of 10,070 vulnerabilities, we find that patches are missing for more than half of the vulnerabilities in the two databases; and patches are inconsistent across the two databases for around 20% of the vulnerabilities. Further, based on a dataset of 1,295 vulnerabilities, we manually locate their accurate patches, and observe that patches are in the form of GitHub commits for around 93% of the vulnerabilities; and multiple patches are developed for about 41% of the vulnerabilities, for which the two databases only have a patch recall of around 50%.

Our Approach. Inspired by our empirical study, we propose an automated approach, Tracer, to address the second problem. Tracer is designed to automatically track patches (in the form of commits) for an OSS vulnerability from multiple knowledge sources (i.e., NVD [41], Debian [12], Red Hat [22] and GitHub). Our key idea is that patch commits are often frequently referenced during the reporting, discussion and resolution of an OSS vulnerability. Tracer works in three steps. First, given the CVE identifier of an OSS vulnerability, it constructs a reference network based on multiple knowledge sources, to model resource references during vulnerability reporting, discussion and resolution. Second, it selects patch commits in the network that have high connectivity and high confidence. Finally, it expands the selected patch commits by searching relevant commits across branches of a repository so as to track patches more completely.

Evaluation. To demonstrate the effectiveness of Tracer, we compare it with three heuristic-based patch tracking approaches and two industrial vulnerability databases on the 1,295 vulnerabilities used in our empirical study. Our evaluation has indicated that i) Tracer can find patches for 58.6% to 273.8% more vulnerabilities than heuristic-based approaches; ii) for the vulnerabilities whose patches are found, Tracer can have a higher patch accuracy by up to 116.8% in F1-score than heuristic-based approaches; and iii) Tracer can complement industrial databases by finding patches completely.

To demonstrate the generality of Tracer, we run it against 3,185 vulnerabilities for which only one of the two industrial vulnerability databases provides their patches and 5,468 vulnerabilities for which none of the two industrial vulnerability databases provides their patches. Our evaluation has shown that Tracer can find patches for 67.7% and 51.5% of the vulnerabilities, and achieve a sampled patch precision of 0.823 and 0.888 and a sampled patch recall of 0.845 and 0.899. Moreover, to evaluate the practical usefulness of Tracer, we conduct a user study with 10 participants. Our evaluation has shown that Tracer can help track patches more accurately and quickly.

Contribution. This work makes the following contributions.

- We conducted a large-scale empirical study to understand the quality and characteristics of patches for OSS vulnerabilities.
- We proposed the first automated approach, named Tracer, to track patches of OSS vulnerabilities from multiple knowledge sources.
- We conducted extensive experiments to demonstrate the effectiveness, generality and practical usefulness of Tracer.

2 AN EMPIRICAL STUDY

We design an empirical study to understand the quality and characteristics of patches for OSS vulnerabilities in vulnerability databases by answering the following research questions.

- **RQ1 Coverage Analysis:** how many OSS vulnerabilities have patches included in vulnerability databases? (Sec. 2.2)
- **RQ2 Consistency Analysis:** how many OSS vulnerabilities have consistent patches across vulnerability databases? (Sec. 2.3)
- **RQ3 Type Analysis:** what are the common patch types for OSS vulnerabilities in vulnerability databases? (Sec. 2.4)
- **RQ4 Cardinality Analysis:** what are the mapping cardinalities between OSS vulnerabilities and their patches? (Sec. 2.5)
- **RQ5 Accuracy Analysis:** how is the patch accuracy of OSS vulnerabilities in vulnerability databases? (Sec. 2.6)

We design **RQ1** to measure the prevalence of missing patches for OSS vulnerabilities in different vulnerability databases. We use **RQ2** to quantify the prevalence of inconsistent patches for OSS vulnerabilities across different vulnerability databases. We leverage **RQ3** and **RQ4** to capture the common patch types and mapping cardinalities between OSS vulnerabilities and their patches. We develop **RQ5** to assess the accuracy of patches for OSS vulnerabilities in different vulnerability databases. In summary, our results from **RQ1**, **RQ2** and **RQ5** aim to assess patch quality from different perspectives and motivate the need for an automated approach to accurately find patches for OSS vulnerabilities, and our results from **RQ3** and **RQ4** aim to capture the characteristics of patches from different perspectives, and inspire the design of our automated patch tracking approach.

2.1 Data Preparation

Vulnerability Database Selection. Official vulnerability databases (e.g., CVE List and NVD) do not provide a "patch" field for each collected vulnerability entry. Industrial vulnerability databases leverage official vulnerability databases as the main vulnerability source, and claim that they collect patches manually or semi-automatically. Therefore, industrial vulnerability databases often provide a "patch" field for their collected vulnerabilities. To enable large-scale empirical study of patches, we focus on industrial vulnerability databases.

Initially, we selected the vulnerability databases from four companies, Black Duck [15], WhiteSource [62], Veracode [60] and Snyk [55]. They provide software composition analysis to identify OSS used in an application and report any OSS vulnerabilities. Hence, we believe they achieve good coverage of OSS vulnerabilities. However, Black Duck does not make the vulnerability database publicly accessible, and WhiteSource does not disclose patches for each vulnerability. We finally selected Veracode's and Snyk's vulnerability databases. Hereafter they are referred to as DB_A and DB_B. We also confirmed with Veracode and Snyk that their public databases are complete.

Breadth Dataset Construction. To broadly quantify missing patches in the two industrial vulnerability databases and inconsistent patches across them (i.e., **RQ1** and **RQ2**), we built a breadth dataset of OSS vulnerabilities by crawling all the OSS vulnerabilities

(a) CVEs (b) CVEs with Patches

Figure 1: Overlap Between Two Databases

from DB_A and DB_B as of April 7, 2020 by their publicly available interfaces. We obtained 8,630 and 5,858 CVEs from DB_A and DB_B. DB_A and DB_B contain a union of 10,070 CVEs.

Depth Dataset Construction. To accurately characterize patch types, mapping cardinalities and patch accuracy (i.e., **RQ3**, **RQ4** and **RQ5**), we built a depth dataset of OSS vulnerabilities, whose size is smaller than the breadth dataset but whose patches are all manually identified to ensure the completeness and accuracy. Specifically, to balance the ease of patch accuracy analysis across two databases and the effort of manual patch identification, we selected 1,417 CVEs for which both databases reported their patches. For each CVE, two of the authors separately found its patches by analyzing patches reported by both databases, looking into CVE description and references in NVD, and searching GitHub repositories and Internet resources. When they had disagreements, a third author was involved into the discussion for consensuses. Finally, they successfully found patches for 1,295 CVEs, while they were still uncertain for 122 CVEs due to limited disclosed information. These 1,295 CVEs mainly cover seven programming languages. Therefore, we believe our depth dataset is representative of OSS vulnerabilities.

2.2 Coverage Analysis (RQ1)

Fig. 1a shows the overlap of CVEs between DB_A and DB_B, and Fig. 1b presents the overlap of CVEs with patches between them. We can see that different vulnerability databases have different coverage of OSS vulnerabilities. DB_A and DB_B have an overlap of 4,418 CVEs, while respectively covering 4,212 and 1,440 unique CVEs. Moreover, 3,607 (41.8%) and 2,412 (41.2%) of the CVEs have their patches provided in DB_A and DB_B. Overall, of all the 10,070 CVEs, 4,602 (45.7%) CVEs have patches provided by at least one vulnerability database. These results indicate that both vulnerability databases have a moderately low patch coverage, and missing patches are prevalent. Automated patch tracking approaches are needed to help find missing patches.

2.3 Consistency Analysis (RQ2)

To analyze patch consistency across the two databases, we focus on CVEs with patches (i.e., Fig. 1b). As a CVE may have a set of patches, we consider two databases as having consistent patches for a CVE if their patch sets are the same. We distinguish patch inconsistency between existence and content inconsistency. The former refers to two cases that one database provides patches for a CVE, but the other database either does not cover the CVE, or covers the CVE but does not provide patches. It reflects the incompleteness of collected OSS vulnerabilities and their patches. The latter refers to two cases that both databases provide patches for a CVE, and their patch sets have an inclusion relationship, or do not have an inclusion relationship but are different from each other. It shows the potential inaccuracy of patches.

Table 1: Patch Consistency and Inconsistency Results

Cons.	Existence Inconsistency			Content Inconsistency		
	Total	No CVE	No Patch	Total	Inclusion	Difference
907 (19.7%)	3,185 (69.2%)	1,392 (30.2%)	1,793 (39.0%)	510 (11.1%)	176 (3.8%)	334 (7.3%)

Table 1 shows our patch consistency analysis results. The first column gives the number of CVEs with consistent patches. The second to fourth columns report the number of CVEs with existence inconsistent patches, and last three columns list the number of CVEs with content inconsistent patches. It can be seen that i) only 907 (19.7%) of the 4,602 CVEs have consistent patches; ii) more than two-thirds (i.e., 3,185 (69.2%)) of the CVEs have existence inconsistency, where 1,392 (30.2%) of the CVEs are not included in DB_A or DB_B, and 1,793 (39.0%) of the CVEs are included but do not have patches in DB_A or DB_B; and iii) 510 (11.1%) of the CVEs incur content inconsistency, where 176 (3.8%) of the CVEs' patches from one database are included in the patches from the other; and 334 (7.3%) of the CVEs have different and non-inclusive patch sets across DB_A and DB_B. These results indicate that these vulnerability databases often report inconsistent patches, the incompleteness of collected OSS vulnerabilities and their patches is severe in these vulnerability databases.

2.4 Type Analysis (RQ3)

We find 3,043 patches for the 1,295 CVEs in our depth dataset by manual analysis. Specifically, 2,852 (93.7%) patches are in the type of GitHub commits potentially due to a wide adoption of GitHub across open source software. 136 (4.5%) patches are in the type of SVN commits potentially due to the prevalence of SVN before the introduction of GitHub, whereas only 55 (1.8%) patches are in the type of commits from other Git platforms. Besides, from the perspective of CVEs, 1,202 (92.8%) of the 1,295 CVEs have the patches in the type of GitHub commits, 4 (0.3%) CVEs have their patches in the type of SVN commits, and 48 (3.7%) CVEs have their patches in the type of both GitHub and SVN commits due to the migration from SVN to GitHub. Only 30 (2.3%) CVEs have some patches in the type of commits from other Git platforms. These results demonstrate that patches for OSS vulnerabilities are mostly in the type of GitHub commits. Thus, patch tracking approaches can specifically focus on GitHub commits.

2.5 Cardinality Analysis (RQ4)

We categorize three types of mapping cardinalities between CVEs and their patches. In detail, 567 (43.8%) of the CVEs have a *one-to-one mapping* to their patch; i.e., they have only one single patch to fix the vulnerability. Hereafter this category is referred to as *SP*.

195 (15.1%) of the CVEs have a *one-to-some mapping* to the patches, meaning that they have multiple *equivalent* patch sets and any one of the patch sets is sufficient to patch the vulnerability. Hereafter we refer to this category as *MEP*. Two patches are equivalent if they have the same code differences. It is mainly caused by two reasons. First, a CVE is patched by a pull request which is merged. Thus, the pull request commits and merged commits are equivalent patch sets for the CVE. Second, the repository of OSS is migrated from SVN to GitHub. Thus, commits for patching a CVE can be in the repository on SVN and GitHub, and SVN commits and GitHub commits are equivalent.

533 (41.2%) of the CVEs have a *one-to-many mapping* to patches, which can be further classified into three types. First, a CVE is fixed

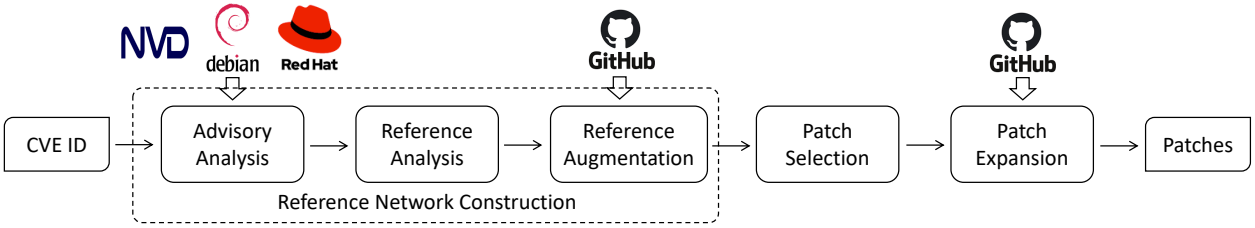

Figure 2: Approach Overview of Tracer

Table 2: Patch Accuracy of Two Databases

Cardinality	Number	DB_A			DB_B		
		Pre.	Rec.	F1	Pre.	Rec.	F1
1:1 (SP)	567	0.908	0.915	0.910	0.900	0.921	0.906
1:*i* (MEP)	195	0.935	0.898	0.902	0.924	0.909	0.906
1:*n* (MP)	101	0.923	0.483	0.616	0.911	0.520	0.638
1:*n* (MB)	372	0.941	0.510	0.620	0.932	0.436	0.555
1:*n* (MR)	60	0.913	0.610	0.695	0.964	0.526	0.636
Total	1,295	0.923	0.748	0.793	0.917	0.730	0.771

by multiple separate commits in a branch. This is because the CVE is difficult to fix or the initial patch is not complete. Hereafter we refer to this type as *MP*, accounting for 101 (7.8%) of the CVEs. Second, a CVE is fixed by multiple patch sets in multiple branches because the CVE affects multiple versions of OSS and each version is maintained on a separate branch. These multiple patch sets should be identified because patches for different versions can be different. Hereafter this type is referred to as *MB*, covering 372 (28.7%) of the CVEs. Third, a CVE is fixed by multiple patch sets in multiple repositories. This is because the CVE affects multiple OSS, or multiple versions of OSS are maintained in separate repositories. Hereafter we refer to this type as *MR*, which covers 60 (4.6%) of the CVEs.

These results demonstrate various mapping cardinalities between CVEs and their patches. They should be considered to ensure completeness when tracking patches for OSS vulnerabilities.

2.6 Accuracy Analysis (RQ5)

We use precision, recall and F1-score as the indicators of patch accuracy. For the CVEs having one-to-some mapping to their patches, we consider reporting one of the multiple equivalent patches as correct. For example, for a CVE that has two equivalent patches, a database that reports one of the two equivalent patches has a full precision and a full recall, while a database that reports one of the two equivalent patches and another irrelevant patch achieves a half precision and a full recall. Table 2 breaks down the accuracy results of the two databases with respect to the mapping cardinalities. The second column reports the number of CVEs in each mapping cardinality, and the next six columns report patch accuracy in the two databases.

DB_A and DB_B achieve a high precision and a high recall of about 90% for the CVEs belonging to *SP* and *MEP*, while having a high precision of above 90% but a low recall of around 50% for the CVEs having one-to-many mappings to their patches (i.e., *MP*, *MB* and *MR*). These results show that these vulnerability databases often miss some patches, especially for CVEs with multiple patches, and such missing information would make it challenging to achieve accurate software composition analysis. It reflects the need to automatically find complete patches for OSS vulnerabilities.

3 OUR APPROACH

Based on the findings from our empirical study, we propose an automated approach, Tracer, to track patches (in the form of commits) for OSS vulnerabilities. The underlying idea of Tracer is that patch commits are often frequently referenced during the reporting, discussion and resolution of an OSS vulnerability in various advisory sources. Fig. 2 presents an overview of Tracer. It takes as an input the CVE identifier of an OSS vulnerability, and returns its patches. Tracer works in three steps. First, it constructs a reference network for the CVE starting from multiple advisory sources (i.e., NVD, Debian [12], Red Hat [22] and GitHub). The goal is to model resource references during the reporting, discussion and resolution of the CVE. Second, it selects the patch nodes (i.e., patch commits) in the network which have high connectivity and high confidence, and thus are most likely to be patches for the CVE. Finally, it expands the selected patch commits via searching relevant commits across branches of the same repository. The goal is to establish a potential one-to-many mapping between the CVE and its patches. In the following subsections, we will elaborate on each step in detail.

3.1 Reference Network Construction

The first step of Tracer consists of three sub-steps. The first two sub-steps (i.e., *advisory analysis* and *reference analysis*) construct a reference network via analyzing advisories from NVD, Debian and Red Hat. The last sub-step (i.e., *reference augmentation*) augments the reference network by searching relevant commit links from GitHub.

Advisory Analysis. Tracer first initializes the reference network by setting the CVE under analysis as the *root node*. It then adds three *advisory source nodes* (i.e., NVD, Debian and Red Hat) as the child node of the root node. These advisory source nodes are used to visualize where the finally selected patches originate.

Example 3.1. Fig. 3 presents the complete reference network for CVE-2017-11428. The top layer shows the root node, and the second layer shows the advisory source nodes.

Then, Tracer respectively requests the advisory from NVD, Debian and Red Hat with the CVE identifier. Specifically, as NVD provides structured data feeds [42] of all vulnerabilities in the form of JSON by year, Tracer requests and parses the corresponding JSON file to obtain the NVD advisory. As Debian stores advisories at a repository [11], Tracer extracts the Debian advisory from the repository. As Red Hat provides WebService API [23], Tracer uses it to retrieve the Red Hat advisory. Notice that Debian tracks all CVEs on NVD, and Red Hat tracks some of them.

Tracer extracts URL references in each requested advisory and adds them as child nodes of the corresponding advisory source node.

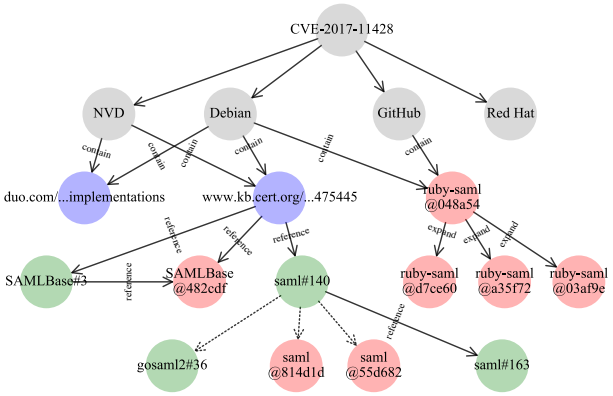

Figure 3: Reference Network for CVE-2017-11428

For an NVD advisory, Tracer extracts URL references in the "references" field, where references to advisories and solutions are listed. Similarly, for a Debian advisory, Tracer extracts URL references in the "Notes" field. For a Red Hat advisory, Tracer uses a regular expression to extract URL references in the "comments" field, where developers discuss and record the resolution process of the vulnerability and may list references to patches.

Example 3.2. As shown in the third layer in Fig. 3, the NVD advisory for CVE-2017-11428 contains two references. One is a reference to a blog that describes the technical detail of this vulnerability, and the other is a reference to a third-party advisory. The two references are also contained by the Debian advisory which further contains a reference to a GitHub commit ruby-saml@048a54 [47] which fixes this vulnerability. Red Hat does not collect this CVE.

Tracer also classifies these reference nodes into three types, i.e., *patch node*, *issue node* and *hybrid node*. We distinguish patch nodes as our goal is to find patches for the CVE. We distinguish issues nodes because issue trackers may assign an issue identifier to the CVE, where developers discuss its resolution and often list references to patches. Reference nodes that are not identified as patch or issue nodes are regarded as hybrid nodes, which can be blogs, third-party advisories, etc. Inspired by our patch type analysis (Sec. 2.4), Tracer identifies a reference node as a patch node if its URL contains "git" and matches a regular expression of commit identifier (i.e., Git platform commits), or contains "svn" and matches a regular expression of commit identifier (i.e., SVN commits). Tracer identifies a reference node as an issue node if its URL contains "/github.com/" and "/issues/" (i.e., GitHub issues), contains "/github.com/" and "/pull/" (i.e., GitHub pull requests), or contains one of the keys "bugzilla", "jira", "issues", "bugs", "tickets" and "tracker" and matches a regular expression of issue identifier (i.e., issues from other issue trackers).

Example 3.3. As shown in the third layer in Fig. 3, the two references contained in both the NVD and Debian advisory are identified as hybrid nodes (i.e., the two purple nodes). The reference that is only contained in the Debian advisory is successfully identified as a patch node (i.e., the red node).

Reference Analysis. For each of the reference nodes constructed in the previous sub-step, Tracer applies the following two analyses to construct the reference network in a layered way.

If the reference node is a patch node, Tracer requests the commit and analyzes whether it only changes test code or non-source code files. If yes, this patch node is removed from the reference network as it cannot be the patch. Tracer identifies test code changes by checking the "test" string in paths of modified files, and identifies non-source code changes by checking the suffix of modified files.

If the reference node is an issue or hybrid node, Tracer first requests the URL and gets the response (i.e., HTML text). Then, it uses a regular expression to extract URL references in plain text, and uses an HTML parser to get URL references in hyperlinks (i.e., <a> tags). These extracted URL references are then checked in the same way as the previous sub-step to identify patch and issue references, which are added as child nodes of the reference node under analysis. No more hybrid references will be added after this layer as the deeper we explore the reference network, the more noise would be introduced by hybrid references. In other words, only the hybrid references directly contained in the NVD, Debian and Red Hat advisories are included in the reference network. There is one exception in the above analysis for URL references to GitHub issues or commits. GitHub issue reports often contain references to issues or commits from other repositories, which can introduce noise to the reference network. To this end, if the reference node under analysis corresponds to a GitHub issue, its extracted URL reference that is not from the same repository will not be added to the reference network.

Tracer repeats the above two analyses on the newly-added nodes until there is no newly-added node or the depth of our reference network reaches a limit (which is 5 by default).

Example 3.4. In the first iteration, Tracer keeps the patch node ruby-saml@048a54 in the third layer in Fig. 3 because it changes non-test source code files. It identifies that one hybrid node in the third layer does not reference to any issue/commit, and the other references to two issues SAMLBase#3 and saml#140 and one commit SAMLBase@482cdf. In the next iteration, it finds that SAMLBase#3 references to SAMLBase@482cdf, and saml#140 references to two issues gosaml2#36 and saml#163 and two commits saml@814d1d and saml@55d682. However, gosaml2#36 is not included in our reference network as it belongs to a different repository than saml#140; and saml@814d1d and saml@55d682 are also not included because they only change test code. Notice that these not included nodes are still shown in Fig. 3 for the ease of presentation, but are connected by dotted arrow lines. After this iteration, the depth limit is reached.

Reference Augmentation. Besides NVD, Debian and Red Hat which are explicit advisory sources, repository hosting platforms can be regarded as an implicit source because patches are often hidden in the commit history. Hence, in this sub-step, Tracer searches repository hosting platforms for patch commits of the CVE in order to further augment the reference network.

Inspired by our patch type analysis (Sec. 2.4), here we only search GitHub as most patches are in the type of GitHub commits. Besides, issues trackers often assign an issue identifier to the CVE. Similarly, advisory publishers usually assign an advisory identifier to the CVE. For example, the vendor advisory of CVE-2019-10426 assigns an advisory identifier of SECURITY-1573 [25], and the issue tracker assigns an issue identifier of THRIFT-4647 [57] to CVE-2018-11798.

Hence, Tracer uses a regular expression to extract issue and advisory identifiers respectively from the URL of issue and hybrid nodes in our reference network constructed in the previous two sub-steps.

Then, Tracer uses the CVE identifier and extracted issue and advisory identifiers as the key to search for commits by REST API [17] provided by GitHub. This API returns up to 1,000 results for a search. To reduce noise, for each returned commit, Tracer checks whether its owner and repository name matches the vendor and product name of any CPE of the CVE. CPE is a structured naming scheme for affected software of the CVE, which can be parsed from the JSON file from NVD. Here we follow Dong et al.'s matching criterion [13] to have the flexibility to handle the slightly different format of the same software name; i.e., two software names are regarded as a match if the number of matched words is not less than the number of unmatched words. Besides, Tracer also checks whether the commit changes non-test source code files. If both checks are passed, Tracer adds it as a child node of the advisory source node of GitHub.

Example 3.5. For CVE-2017-11428, Tracer fails to extract any issue or advisory identifier. Thus, it uses the CVE identifier to search for GitHub commits. The matched commit is ruby-saml@048a54, and its owner and repository name is "onelogin" and "ruby-saml". As the vendor and product name in the CPE of this CVE is "onelogin" and "ruby-saml", a complete match is achieved. As this commit is already included in our reference network, Tracer connects it as a child node of the new advisory source node of GitHub, as shown in Fig. 3.

3.2 Patch Selection

The second step of Tracer is to select patches from our reference network for the CVE under analysis accurately and completely. To this end, we use two heuristics, and combine their selected patches.

Confidence. We directly select the patch nodes that we treat as having high confidence of being the correct patch for the CVE under analysis. Specifically, we consider two kinds of patch nodes in our reference network as having such high confidence. First, the patch nodes that are a direct child node of the advisory source node of NVD are considered as having high confidence. The reason is that NVD is established with a strong community effort, each vulnerability is manually confirmed with several procedures, and the data can be continuously updated after the initial vulnerability reporting. Second, the patch nodes that are a direct child node of the advisory source node of GitHub are considered as having high confidence. The reason is that the way Tracer adds such patch nodes ensures that the commit message contains the CVE identifier, advisory identifier, or issue identifier of the CVE and the name of its belonging owner and repository matches the vendor and product name of the CPE.

Example 3.6. From the reference network for CVE-2017-11428 in Fig. 3, Tracer directly selects the patch node ruby-saml@048a54 because it is a child node of the advisory source node of GitHub and is considered as having high confidence of being the correct patch for CVE-2017-11428. In fact, this commit is one of the correct patches.

Connectivity. The confidence-based heuristic alone is often not strong enough to locate patches accurately and completely because NVD may not contain patch references, and CVE identifier, advisory identifier or issue identifier of a CVE might not be contained in commit messages. Hence, inspired by the idea that a correct patch would be frequently referenced during the reporting, discussion, and resolution of a CVE in various advisory sources (i.e., the patch node would be widely connected to the root node in our reference network), we further design a connectivity-based heuristic.

Specifically, we measure the connectivity of a patch node to the root node in our reference network based on two dimensions. First, the more paths the root node can reach a patch node, the higher connectivity to the root node the patch node has. Second, the shorter the paths from the root node to a patch node, the higher connectivity to the root node the patch node has. To combine these two dimensions, we use Eq. 1 to compute the connectivity of a patch node to the root node, where $p = 1, ..., n$ denotes one of the n paths from the root node to the patch node, and d_p denotes the length of a path p. Considering the high confidence of the two advisory sources of NVD and GitHub, the length of a path is changed by a decrease of 1 if the path originates from the advisory source node of NVD and GitHub.

$$connectivity = \sum_{p=1}^{n} \frac{1}{2^{(d_p-1)}} \tag{1}$$

Based on the connectivity of each patch node to the root node, Tracer selects the patch nodes with the highest connectivity.

Example 3.7. In Fig. 3, there are two paths from root node to the patch node ruby-saml@048a54. One originates from Debian with a length of 2, and has connectivity of 0.5. The other originates from GitHub with an original length of 2 and a changed length of 1, and has connectivity of 1. Thus, the connectivity of ruby-saml@048a54 to the root node is 1.5. Similarly, there exist four paths from the root node to the patch node SAMLBase@482cdf, respectively having connectivity of 0.5, 0.25, 0.25, and 0.125. Hence, the connectivity of SAMLBase@482cdf to the root node is 1.125. Tracer selects ruby-saml@048a54 as the patch as it has the highest connectivity.

3.3 Patch Expansion

The third step of Tracer is to expand the patches selected in the second step by searching relevant commits across branches of the same repository. It is inspired by our cardinality analysis (Sec. 2.5) where we find more than 40% of the CVEs have a one-to-many mapping to their patches, and these multiple patches often locate in one branch of a repository (as a CVE is difficult to fix or the first patch is not complete) or multiple branches of a repository (as a CVE affects multiple versions whose branches are separately maintained). For these multiple patches, our reference network constructed in the previous two steps often does not capture them completely. Besides, our patch type analysis (Sec. 2.4) shows that most patches are in the type of GitHub commits. Therefore, the third step of Tracer is designed as follows: for each selected patch that is in the type of a GitHub commit, Tracer locates its repository, collects all branches in this repository, and searches the commits within a specific span of each branch for commits that are potentially patches for the CVE under analysis.

Specifically, for a selected patch that is in the type of a GitHub commit, Tracer uses a regular expression to extract the owner and repository information from the patch URL, owing to the well-structured commit URL in GitHub. Based on the owner and repository data, Tracer retrieves all branches in the repository by GitHub's REST API [18]. Then, for each branch, Tracer retrieves the commits created before and after the selected patch within a specific span (which

is 30 days by default) by GitHub's REST API [19]. We do not retrieve all the commits for balancing performance and accuracy. Then, for each retrieved commit, Tracer uses the following two criteria to determine whether the commit is the patch for the CVE under analysis: i) the commit message of the retrieved commit is the same as, contains, or is contained by the commit message of the selected patch; or ii) the commit message of the retrieved commit contains the CVE identifier, advisory identifier or issue identifier. If a retrieved commit satisfies one of the two criteria, Tracer also adds such expanded patches as child nodes of the selected patch.

Finally, Tracer returns the selected patches in the second step and the expanded patches in the third step as the patches for the CVE under analysis. Besides, our reference network is also returned for the ease of confirming returned patches.

Example 3.8. For the selected patch ruby-saml@048a54 (locating on the master branch) for CVE-2017-11428, Tracer expands it by finding three commits ruby-saml@d7ce60 [48], ruby-saml@a35f72 [49] and ruby-saml@03af9e [50] which have the same commit message to ruby-saml@048a54 but respectively locate on branches 0.8.3 – 0.8.17, v0.9.3 and v1.6.2. As shown in Fig. 3, Tracer adds them as child nodes of ruby-saml@048a54. Notice that these four patches are all correct and involve different code changes. The two vulnerability databases in Sec. 2 only report the patch ruby-saml@048a54.

4 EVALUATION

Research Questions. We design our evaluation to answer the following four research questions.

- **RQ6 Effectiveness Evaluation:** how is the effectiveness of Tracer in tracking patches, compared to existing heuristic-based approaches and two industrial vulnerability databases? (Sec. 4.1)
- **RQ7 Ablation Analysis:** how is the contribution of each component in Tracer to its achieved effectiveness? (Sec. 4.2)
- **RQ8 Generality Evaluation:** how is the generality of Tracer to OSS vulnerabilities beyond our depth dataset? (Sec. 4.3)
- **RQ9 Usefulness Evaluation:** how is the usefulness of Tracer in practice? (Sec. 4.4)

We use our depth dataset to answer **RQ6** and **RQ7**, and build two datasets to answer **RQ8**. We conduct a user study to answer **RQ9**.

Evaluation Metrics. We use four metrics to measure the effectiveness of patch tracking in **RQ6**, **RQ7** and **RQ8**. The first metric is the number of CVEs whose patches are not found by a patch tracking approach. It measures *patch coverage* by only considering whether patches are found or not. The other metrics are precision, recall and F1-score (i.e., the same metrics in Sec. 2.6), which measure *patch accuracy* for those CVEs whose patches are found by a patch tracking approach. In **RQ9**, we use patch accuracy and the time consumed by users with/without the help of Tracer in finding patches.

4.1 Effectiveness Evaluation (RQ6)

Comparison to Heuristic-Based Approaches. We pick two widely used heuristics: i) searching NVD references of a CVE for commits (e.g., [14, 33, 34]) and ii) searching GitHub commit history for commits containing the identifier of a CVE in commit messages (e.g., [61, 67]). The first heuristic can be used to approximate the quality of hidden patches in NVD. In fact, we manually track the hidden patches in NVD, which achieves similar effectiveness results to this heuristic. The second heuristic is usually used for searching patches for a known OSS, we adapt it to search patches for a CVE by further checking whether the owner and repository match the vendor and product in CPE (i.e., our strategy in reference augmentation). We also investigate a third heuristic that combines the results of the above two heuristics. Table 3 and 4 respectively present the effectiveness results of the three heuristics and Tracer (and its variants, which will be discussed in Sec. 4.2).

On one hand, for patch coverage, all the three heuristics fail to find any patch (i.e., return nothing) for a very large part (i.e., 59.3%, 76.4% and 44.5%) of the CVEs across all mapping cardinalities. Differently, Tracer fails to find a patch for only 12.0% of the CVEs. On the other hand, for patch accuracy on the CVEs whose patches are found, the first heuristic achieves a high patch precision due to the high confidence of NVD references, but a low patch recall on the CVEs with one-to-many mappings; the second heuristic has both a low patch precision and a low patch recall; and the third heuristic achieves a patch precision and a patch recall between the first and second heuristic. Tracer has a lower patch precision, a higher patch recall (a significantly higher patch recall on CVEs belonging to *MP* and *MB*), and a comparable F1-score than the first heuristic. We believe it is acceptable because Tracer finds patches for 116.3% more CVEs for which the first heuristic fails to find any patch. Besides, Tracer improves the second and third heuristic in F1-score by 116.8% and 16.3%.

> Existing heuristics fail to find any patch for a very large part of vulnerabilities, while Tracer finds patches for 58.6% to 273.8% more vulnerabilities than them. For the vulnerabilities whose patches are found, Tracer has either a comparable F1-score or a higher F1-score by up to 116.8% than existing heuristics.

Comparison to Vulnerability Databases. We are not aware of how much manual effort or what automated approach is involved in the construction of industrial vulnerability databases. Some of them claimed that they collect patches manually or semi-automatically. Therefore, the goal of our comparison to industrial databases is not to demonstrate the superiority or inferiority of Tracer over industrial databases, but to assess the level of effectiveness Tracer can achieve and the worthiness of Tracer, and explore whether Tracer can potentially improve or complement existing industrial databases.

As shown in Table 2 and 4, Tracer finds patches for 12.0% fewer CVEs than DB_A and DB_B. This is determined by the way we construct this used depth dataset (i.e., our depth dataset includes vulnerabilities whose patches are provided by DB_A and DB_B). We will demonstrate in Sec. 4.3 how Tracer works on vulnerabilities whose patches are not provided by DB_A and DB_B.

On the CVEs whose patches are found, Tracer has a 6.4% and 5.8% lower patch precision than DB_A and DB_B. This might be potentially due to the manual effort involved in industrial database construction. Besides, Tracer has a 15.5% and 18.4% higher patch recall than DB_A and DB_B (especially on CVEs belonging to one-to-many mappings), resulting in a 5.5% and 8.6% higher F1-score than DB_A and DB_B. These results indicate that Tracer is worthwhile with a significantly higher patch recall at the price of a moderately lower patch precision. Tracer has the merit to complement industrial databases by reducing manual effort and tracking patches completely.

Table 3: Effectiveness of Existing Heuristic-Based Approaches

Cardinality	Number	Searching NVD References				Searching GitHub Commit History				Searching NVD and GitHub			
		Not Found	Pre.	Rec.	F1	Not Found	Pre.	Rec.	F1	Not Found	Pre.	Rec.	F1
1:1 (SP)	567	285 (50.3%)	0.973	0.986	0.977	472 (83.2%)	0.416	0.642	0.471	222 (39.2%)	0.839	0.930	0.864
1:*i* (MEP)	195	125 (64.1%)	0.932	0.925	0.921	162 (83.1%)	0.472	0.490	0.452	104 (53.3%)	0.821	0.867	0.820
1:*n* (MP)	101	68 (67.3%)	0.980	0.552	0.683	73 (72.3%)	0.536	0.445	0.461	52 (51.5%)	0.779	0.605	0.647
1:*n* (MB)	372	244 (65.6%)	0.979	0.416	0.546	246 (66.1%)	0.445	0.236	0.284	171 (46.0%)	0.704	0.393	0.465
1:*n* (MR)	60	46 (76.7%)	1.000	0.708	0.794	37 (61.7%)	0.627	0.345	0.413	27 (45.0%)	0.801	0.539	0.604
Total	1,295	768 (59.3%)	0.970	0.805	0.842	990 (76.4%)	0.461	0.417	0.386	576 (44.5%)	0.793	0.732	0.720

Table 4: Contribution of Each Component in Tracer

Cardinality	Number	Tracer				v_1^1: Tracer w/o NVD				v_1^2: Tracer w/o Debian			
		Not Found	Pre.	Rec.	F1	Not Found	Pre.	Rec.	F1	Not Found	Pre.	Rec.	F1
1:1 (SP)	567	102 (18.0%)	0.860	0.951	0.881	286 (50.4%)	0.820	0.936	0.846	110 (19.4%)	0.847	0.943	0.869
1:*i* (MEP)	195	6 (3.1%)	0.886	0.918	0.888	79 (40.5%)	0.882	0.935	0.886	8 (4.1%)	0.880	0.912	0.882
1:*n* (MP)	101	20 (19.8%)	0.872	0.741	0.761	41 (40.6%)	0.881	0.728	0.766	22 (21.8%)	0.851	0.716	0.739
1:*n* (MB)	372	23 (6.2%)	0.861	0.788	0.795	84 (22.6%)	0.876	0.780	0.800	28 (7.5%)	0.838	0.760	0.771
1:*n* (MR)	60	4 (6.7%)	0.831	0.620	0.659	8 (13.3%)	0.848	0.551	0.624	5 (8.3%)	0.819	0.613	0.651
Total	1,295	155 (12.0%)	0.864	0.864	0.837	498 (38.5%)	0.856	0.839	0.815	173 (13.4%)	0.848	0.849	0.821
Cardinality	Number	v_1^3: Tracer w/o Red Hat				v_1^4: Tracer w/o GitHub				v_1^5: Tracer w/o Network			
		Not Found	Pre.	Rec.	F1	Not Found	Pre.	Rec.	F1	Not Found	Pre.	Rec.	F1
1:1 (SP)	567	113 (19.9%)	0.853	0.943	0.874	149 (26.3%)	0.898	0.943	0.908	177 (31.2%)	0.910	0.972	0.925
1:*i* (MEP)	195	7 (3.6%)	0.883	0.918	0.886	19 (9.7%)	0.887	0.921	0.892	78 (40.0%)	0.956	0.959	0.941
1:*n* (MP)	101	21 (20.8%)	0.880	0.736	0.760	28 (27.7%)	0.873	0.690	0.726	40 (39.6%)	0.943	0.669	0.743
1:*n* (MB)	372	35 (9.4%)	0.844	0.761	0.767	39 (10.5%)	0.874	0.752	0.773	109 (29.3%)	0.908	0.575	0.659
1:*n* (MR)	60	4 (6.7%)	0.738	0.640	0.618	7 (11.7%)	0.816	0.545	0.604	10 (16.7%)	0.920	0.641	0.712
Total	1,295	180 (13.9%)	0.851	0.853	0.823	242 (18.7%)	0.883	0.841	0.835	414 (32.0%)	0.918	0.812	0.823
Cardinality	Number	v_2^1: Tracer w/o Selection				v_2^2: Tracer w/o Connectivity				v_2^3: Tracer w/o Confidence			
		Not Found	Pre.	Rec.	F1	Not Found	Pre.	Rec.	F1	Not Found	Pre.	Rec.	F1
1:1 (SP)	567	102 (18.0%)	0.632	0.961	0.680	245 (43.2%)	0.892	0.978	0.913	102 (18.0%)	0.860	0.942	0.879
1:*i* (MEP)	195	6 (3.1%)	0.622	0.976	0.682	111 (56.9%)	0.929	0.939	0.915	6 (3.1%)	0.888	0.913	0.889
1:*n* (MP)	101	20 (19.8%)	0.615	0.933	0.656	56 (55.4%)	0.953	0.685	0.764	20 (19.8%)	0.880	0.722	0.751
1:*n* (MB)	372	23 (6.2%)	0.616	0.903	0.657	191 (51.3%)	0.927	0.787	0.821	23 (6.2%)	0.871	0.765	0.784
1:*n* (MR)	60	4 (6.7%)	0.368	0.891	0.394	27 (45.0%)	0.885	0.722	0.772	4 (6.7%)	0.849	0.462	0.550
Total	1,295	155 (12.0%)	0.611	0.940	0.658	630 (48.6%)	0.910	0.889	0.871	155 (12.0%)	0.869	0.844	0.826
Cardinality	Number	v_2^4: Tracer with Path Length				v_2^5: Tracer with Path Number				v_3: Tracer w/o Expansion			
		Not Found	Pre.	Rec.	F1	Not Found	Pre.	Rec.	F1	Not Found	Pre.	Rec.	F1
1:1 (SP)	567	102 (18.0%)	0.833	0.957	0.859	102 (18.0%)	0.805	0.951	0.837	102 (18.0%)	0.871	0.948	0.889
1:*i* (MEP)	195	6 (3.1%)	0.848	0.945	0.867	6 (3.1%)	0.849	0.920	0.858	6 (3.1%)	0.910	0.914	0.902
1:*n* (MP)	101	20 (19.8%)	0.849	0.760	0.742	20 (19.8%)	0.801	0.756	0.726	20 (19.8%)	0.873	0.696	0.732
1:*n* (MB)	372	23 (6.2%)	0.830	0.798	0.770	23 (6.2%)	0.833	0.811	0.791	23 (6.2%)	0.860	0.506	0.590
1:*n* (MR)	60	4 (6.7%)	0.652	0.747	0.590	4 (6.7%)	0.789	0.630	0.644	4 (6.7%)	0.847	0.567	0.629
Total	1,295	155 (12.0%)	0.827	0.882	0.812	155 (12.0%)	0.819	0.873	0.809	155 (12.0%)	0.873	0.771	0.776

Tracer achieves a 15.5% to 18.4% higher patch recall and a 5.5% to 8.6% higher F1-score than the two industrial vulnerability databases, while sacrificing up to 6.4% lower patch precision. Tracer can complement industrial vulnerability databases by reducing manual effort and tracking patches completely.

False Negative Analysis. We manually analyze the CVEs for which Tracer finds no patch or misses some of the patches, and summarize five main reasons. First, for some old CVEs, the references contained in NVD, Debian and Red Hat are limited, and some of them even become invalid. As a result, Tracer fails to construct a complete reference network. Second, some key references (e.g., issue reports) about a CVE are missing from NVD, Debian and Red Hat. As a result, Tracer fails to be directed to the correct patch. For example, for CVE-2018-14642, its issue report [58] is not contained in any of the three advisory sources. However, following the issue report, we could find the patch [59]. Third, the commit message of a patch has semantic similarity to the CVE description, but does not contain the CVE identifier. Hence, our reference augmentation fails to catch it. For example, for CVE-2019-10077 [40], our patch commit [30] fixes it without indicating the CVE identifier. Fourth, GitHub's REST API for commit search returns 1,000 results, which may miss the correct patch commit in our reference augmentation. Fifth, only one patch with the highest connectivity is selected in our patch selection. Thus, correct patches for CVEs belonging to one-to-many mappings might be missed although they are already included in our reference network.

False Positive Analysis. We also manually analyze the CVEs for which Tracer finds incorrect patches, and summarize two main reasons. First, the commit that introduces the CVE is referenced during the discussion and resolution of the CVE. As a result, Tracer

falsely identifies it as a patch commit due to the lack of semantic understanding of the context where the commit is referenced. For example, for CVE-2020-5249, the commit that introduces the CVE [20] and the commit that fixes the CVE [21] are referenced in the same comment of the issue report. Second, multiple CVEs and their issues and patches are listed on the same page. As a result, patches for other CVEs might be falsely identified by TRACER due to the lack of semantic understanding. For example, for CVE-2018-15750, its patches are maintained in the release note [52] with CVE-2018-15751, and the release note is all referenced by NVD, Debian, and Red Hat.

The five and two reasons for false negatives and false positives are respectively summarized by manual analysis, which can be leveraged to further improve the effectiveness of TRACER.

4.2 Ablation Analysis (RQ7)

Table 4 presents the results of our ablation study to measure the contribution of various settings in TRACER to its achieved effectiveness.

Removing an Advisory Source. We remove one of the four advisory sources NVD, Debian, Red Hat and GitHub from the first step of TRACER, and generate variants v_1^1, v_1^2, v_1^3 and v_1^4 of TRACER. These four variants suffer an increase in the number of CVEs they fail to find any patch for. v_1^1, v_1^2, v_1^3 and v_1^4 respectively find patches for 30.1%, 1.6%, 2.2% and 7.6% fewer CVEs than TRACER, while achieving comparable precision, recall and F1-score on CVEs they find patches for. These results indicate that all the four advisory sources contribute to finding patches for more CVEs by constructing a more complete reference network, while NVD and GitHub contribute the most.

Removing Reference Network. We do not construct the reference network in a layered way but simply use the direct references contained in the four advisory sources (i.e., we skip reference analysis in the first step of TRACER), which is the variant v_1^5 in Table 4. v_1^5 also suffers an increase in the number of CVEs it fails to find any patch for. v_1^5 finds patches for 22.7% fewer CVEs than TRACER, while having a 6.3% higher precision, a 6.0% lower recall, and a comparable F1-score on CVEs it finds patches for. These results demonstrate that patches are not always directly referenced in NVD, Debian and Red Hat, but might be hidden in indirect references, and our reference network contributes to tracking such hidden patches at the price of an acceptable decrease in precision.

Removing Patch Selection. We do not select some patches in the second step of TRACER but select all the patches in our reference network, which is variant v_2^1 in Table 4. v_2^1 significantly improves TRACER in recall by 8.8%, especially for CVEs belonging to one-to-many mappings, while suffering a large degradation in precision by 29.3% and in F1-score by 21.4% across all cardinalities. These results indicate that our reference network indeed contains most of the correct patches, and our patch selection heuristics contribute to achieving a balance between precision and recall.

Removing Connectivity or Confidence. We remove one of the two heuristics adopted in the second step of TRACER, and generate two variants v_2^2 and v_2^3. Without our connectivity-based heuristic, v_2^2 finds patches for 41.7% fewer CVEs than TRACER, while achieving a 5.3% higher precision, a 2.9% higher recall and a 4.1% higher F1-score. These results indicate that our connectivity-based heuristic contributes to finding patches for more CVEs while introducing acceptable noise. Without our confidence-based heuristic, v_2^3 suffers a slight decrease in recall and F1-score, especially for CVEs belonging to *MR*. These results show that our confidence-based heuristic contributes to achieving a balanced accuracy across all cardinalities.

Reducing Connectivity. We reduce connectivity-based heuristic by only considering path length (i.e., selecting the patch with the shortest path to the root node) and by only considering path number (i.e., selecting the patch with the largest number of paths to the root node), and respectively generate the variant v_2^4 and v_2^5. Both v_2^4 and v_2^5 suffer a 4.3% and 5.2% decrease in precision, a 2.1% and 1.0% increase in recall, and a 3.0% and 3.3% decrease in F1-score. These results demonstrate that our connectivity-based heuristic contributes to improving the precision of TRACER by comprehensively considering the path length and path number of a patch to the root node.

Removing Patch Expansion. We do not expand patches in the third step of TRACER, which is variant v_3 in Table 4. v_3 suffers degradation in recall and F1-score by 10.8% and 7.3%, especially for CVEs with one-to-many mappings. These results show that our patch expansion contributes to finding multiple patches more completely.

Our used advisory sources, reference network, patch selection, and patch expansion all contribute positively to the achieved effectiveness of TRACER in tracking patches.

4.3 Generality Evaluation (RQ8)

To evaluate the generality of TRACER (i.e., whether TRACER is overfitted to our depth dataset), we collect two new vulnerability datasets, and run TRACER against them. The first dataset includes the CVEs for which only one of the two industrial vulnerability databases reports their patches (Fig. 1b), which has 3,185 CVEs. The second dataset includes the CVEs for which none of the two industrial vulnerability databases reports their patches (Fig. 1), which has 5,468 CVEs.

TRACER finds patches for 2,155 (67.7%) of the 3,185 CVEs in the first dataset, and the two industrial vulnerability databases DB_A and DB_B report patches for 2,190 (68.8%) and 995 (31.2%) CVEs. Of the 2,155 CVEs TRACER finds patches for, DB_A and DB_B report patches for 1,455 and 700 CVEs. In addition, TRACER finds patches for 2,816 (51.5%) of the 5,468 CVEs in the second dataset, where DB_A and DB_B report no patch. These results indicate that TRACER complements industrial vulnerability databases by tracking patches for CVEs whose patches are not provided by industrial vulnerability databases.

Then, we respectively sample 100 CVEs TRACER finds patches for from the first and second datasets, and manually find their patches in the same procedure in Sec. 2.1 to measure the accuracy of TRACER. The results are reported in Table 5. Our manual analysis has 9 and 11 uncertain CVEs due to limited disclosed information. Of the 91 CVEs in the first dataset, TRACER has an F1-score of 0.784, while DB_A provides patches for 62 (68.1%) CVEs with a higher F1-score and DB_B provides patches for 29 (31.9%) CVEs with a lower F1-score. Similar to the results in Sec. 4.1, industrial vulnerability databases achieve a higher precision but a lower recall. Of the 89 CVEs in the second dataset, DB_A and DB_B report no patch, whereas TRACER achieves an F1-score of 0.867. These results indicate that TRACER can be generalized to vulnerabilities beyond the ones in our depth dataset.

Table 5: Generality of Tracer over Two New Datasets

Dataset	Number	Tracer			DB_A				DB_B			
		Pre.	Rec.	F1	Not Found	Pre.	Rec.	F1	Not Found	Pre.	Rec.	F1
First Dataset	91	0.823	0.845	0.784	29 (31.9%)	0.935	0.827	0.858	62 (68.1%)	0.885	0.664	0.725
Second Dataset	89	0.888	0.899	0.867	–	–	–	–	–	–	–	–

Table 6: Comparison Results of the Time and Accuracy of 10 Tasks

Approach	All 10 Tasks				5 Single-Patch Tasks				5 Multiple-Patches Tasks			
	Time (mins)	Pre.	Rec.	F1	Time (mins)	Pre.	Rec.	F1	Time (mins)	Pre.	Rec.	F1
w/o Tracer	5.66	0.880	0.677	0.765	5.60	0.960	0.960	0.960	5.72	0.800	0.393	0.527
with Tracer	4.66	0.983	0.920	0.951	3.84	1.000	1.000	1.000	5.48	0.967	0.840	0.899

In addition, we find that the two industrial vulnerability databases have been updated since April 7th, 2020 (i.e., our crawling date). Thus, we investigate the generality of Tracer from another perspective, i.e., how patches tracked by Tracer are similar to their updates. To this end, we update all vulnerability entries in DB_A and DB_B to DB_A^+ and DB_B^+ at March 8th, 2022. Of the CVEs whose patches are found by Tracer but not provided by DB_A (resp. DB_B), 147 (resp. 669) CVEs' patches are newly provided by DB_A^+ (resp. DB_B^+). For 56 (38.1%) (resp. 405 (60.5%)) of the 147 (resp. 669) CVEs, Tracer finds the same patches to the patches provided by DB_A^+ (resp. DB_B^+). For 37 (25.2%) (resp. 199 (29.7%)) of the 147 (resp. 669) CVEs, the patches provided by DB_A^+ (resp. DB_B^+) are included in the patches tracked by Tracer. These results indicate that the patches found by Tracer have a high chance of being accepted by industrial vulnerability databases.

> Tracer finds patches for 67.7% and 51.5% of the CVEs in the two new datasets with a sampled patch precision of 0.823 and 0.888 and a sampled patch recall of 0.845 and 0.899. Tracer is generalizable and complements industrial databases.

4.4 Usefulness Evaluation (RQ9)

In practice, to ensure patch accuracy, security experts still need to verify patches even when automatic tools are used to find patches. To evaluate the usefulness of Tracer in such a usage scenario, we conduct a user study with 10 participants who are required to find patches for 10 CVEs with and without the help of Tracer. We recruit 10 participants from security laboratories in multiple universities and high-tech companies. They are Postdocs, PhD students, master researchers, and engineers majoring in software security. We randomly select 10 CVEs from our depth dataset as tasks. 2 CVEs belong to *SP*, 3 CVEs belong to *MEP*, 1 CVE belongs to *MP*, and 4 CVEs belong to *MB*. To have a fair comparison, we divide participants into two groups (i.e., A and B). Group A is required to complete the first five tasks without Tracer (but can use existing heuristics) and finish the remaining tasks with Tracer. Group B is required to complete the first five tasks with Tracer, and finish the remaining tasks without Tracer.

Table 6 reports the average time consumption and patch accuracy of the 10 tasks. We categorize the 5 CVEs belonging to *SP* and *MEP* as *Single-Patch* tasks, and categorize the 5 CVEs belonging to *MP* and *MB* as *Multiple-Patches* tasks. Overall, with the assistance of Tracer, the participants save time by 17.7% for each task, and improve the patch accuracy in terms of precision, recall and F1-score by 11.7%, 35.9% and 24.3%. In particular, the time saving is significant for the 5 *Single-Patch* tasks, but not significant for the 5 *Multiple-Patches* tasks. This is because the reference network and patches returned by Tracer for *Multiple-Patches* tasks are more complex than those of *Single-Patch* tasks, and participants spend more time understanding the reference network and verify patches. Moreover, the accuracy improvement is significant for the 5 *Multiple-Patches* tasks, but not significant for the 5 *Single-Patch* tasks. In that sense, Tracer is especially useful for CVEs with multiple patches.

We interview each participant to get their feedback about Tracer. Overall, they all appreciate the value of our reference network as it summarizes information from multiple sources, and the different kinds of nodes and relationships in it are helpful to localize and verify patches. As commented by a security engineer from a high-tech company, *"the network graph, as a chain of evidence, is helpful to localize and verify patches"*. Moreover, they suggest including more information for the nodes (e.g., commit message and code differences) instead of a clickable link for the ease of manual review, and also suggest adding more advisory sources. It is worth mentioning that Tracer has been deployed to two high-tech companies that participate this user study. Unfortunately, we cannot disclose its usage statistics due to confidentiality agreement.

> Tracer is useful in practice for security experts to localize patches more accurately and quickly.

4.5 Discussion

Limitations. First, Tracer only uses NVD, Debian and Red Hat as the advisory sources in the step of advisory analysis. However, it is designed to easily leverage other sources (e.g., SecurityFocus [53]) thanks to our lightweight reference analysis. Second, as indicated by our false negative and false positive analysis, the lack of semantic analysis of patches and vulnerabilities hurts the accuracy of Tracer. We plan to utilize semantics in vulnerabilities (e.g., descriptions) and patches (e.g, changed code and the context of patch references) to enhance patch selection and expansion. Third, Tracer depends on the quality of the existing references in advisory sources. If there are not many high-quality references, Tracer might be less effective. Thus, we use multiple advisory sources to reduce this possibility.

Significance. Tracer can benefit security community, academia, and industry by enabling automated patch tracking. For security community, Tracer can notify NVD for missing or incomplete patches for CVEs to enhance CVE information quality and accelerate entry update, and thus benefit the audience of NVD. For academia, Tracer can enable data-driven security analysis (e.g., learning-based vulnerability detection [34, 70]) and empirical studies by providing large-scale patches. For industry, Tracer can assist security engineers in enhancing patch coverage and accuracy of industrial vulnerability

databases, and hence improve the accuracy of software composition analysis (i.e., determining whether vulnerabilities in patched methods in a used OSS are reachable in an application).

5 RELATED WORK

CVE Information Quality. Nguyen and Massacci [39] uncover the unreliability of the vulnerable version data in NVD. To improve its reliability, Nguyen et al. [38] and Dashevskyi et al. [10] develop tools to determine whether older versions are affected by a newly disclosed vulnerability. Dong et al. [13] identify vulnerable software names and versions from vulnerability reports, and find that vulnerability databases miss truly vulnerable versions or falsely include non-vulnerable versions. Chen et al. [5] identify open-source libraries affected by a vulnerability. Chaparro et al. [4] detect the absence of reproduction steps and expected behavior in vulnerability descriptions. Mu et al. [36] show the prevalence of missing reproduction information in vulnerability reports. Jo et al. [29] identify semantic inconsistencies within the cybersecurity domain. These works are focused on different aspects of vulnerability information. Following this direction, our work is focused on the patch of a vulnerability.

A closely related work is from Tan et al. [56]. They use a learning-to-rank algorithm to rank commits in a repository so that patch commits to a CVE are ranked in top positions. However, they make two assumptions: i) the repository of the affected software of a CVE is known, which is not practical and requires manual efforts; and ii) a CVE has a one-to-one mapping to its patches, which does not always hold (Sec. 2.5). Instead, Tracer has no such assumptions.

Patch Analysis. There are many patch analysis tasks to improve security, e.g., patch generation and deployment [14, 37, 66], patch presence testing [9, 26, 69] and secret patch identification [6, 51, 65, 71]. Datasets of security patches have been built for Java [46], C/C++ [16] and specific open-source projects [27]. With such datasets, empirical studies have been conducted to characterize vulnerabilities and their patches [1, 32, 35, 68]. In these works, patches are identified by manual efforts [6, 9, 26, 46, 51, 65, 66, 68, 71] or by heuristic rules like looking for commits in CVE references [14, 16, 27, 32, 35] and searching for CVE identifiers in commits [1, 16, 27]. Such heuristics only search direct references, but patches can be hidden in indirect references. Tracer addresses it by a reference network.

Patch Applications. Patches can be leveraged to enable various security applications, e.g., generating exploits based on patches [2, 67], conducting software composition analysis to determine whether vulnerabilities in a library are reachable through which call paths [43–45, 61], and detecting vulnerabilities by learning vulnerability features [28, 33, 34, 70], by matching vulnerability signatures [24, 31] and by matching both vulnerability and patch signatures [7, 63, 64]. Similar to those patch analysis works, the mappings between CVEs and their patches in these works are mostly identified by manual efforts [43–45, 63] and heuristics rules [28, 33, 34, 61, 67], or directly taken from security advisories that establish the mapping between CVEs and patches for specific projects [24, 31, 64].

6 CONCLUSIONS

We have conducted an empirical study to understand the quality and characteristics of patches for OSS vulnerabilities in two industrial vulnerability databases. We have proposed Tracer to track patches for OSS vulnerabilities. Our extensive evaluation has demonstrated the effectiveness, generality and usefulness of Tracer. We have released the code and data at https://patch-tracer.github.io.

ACKNOWLEDGMENTS

This work was supported by the National Natural Science Foundation of China (Grant No. 61972098).

REFERENCES

[1] Gábor Antal, Márton Keleti, and Péter Hegedűs. 2020. Exploring the Security Awareness of the Python and JavaScript Open Source Communities. In *Proceedings of the 17th International Conference on Mining Software Repositories*. 16–20.
[2] David Brumley, Pongsin Poosankam, Dawn Song, and Jiang Zheng. 2008. Automatic patch-based exploit generation is possible: Techniques and implications. In *Proceedings of the IEEE Symposium on Security and Privacy*. 143–157.
[3] Synopsys Cybersecurity Research Center. 2021. *Open Source Security and Risk Analysis Report*. Retrieved August 31, 2021 from https://www.synopsys.com/content/dam/synopsys/sig-assets/reports/rep-ossra-2021.pdf
[4] Oscar Chaparro, Jing Lu, Fiorella Zampetti, Laura Moreno, Massimiliano Di Penta, Andrian Marcus, Gabriele Bavota, and Vincent Ng. 2017. Detecting missing information in bug descriptions. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. 396–407.
[5] Yang Chen, Andrew E Santosa, Asankhaya Sharma, and David Lo. 2020. Automated identification of libraries from vulnerability data. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Software Engineering in Practice*. 90–99.
[6] Yang Chen, Andrew E Santosa, Ang Ming Yi, Abhishek Sharma, Asankhaya Sharma, and David Lo. 2020. A Machine Learning Approach for Vulnerability Curation. In *Proceedings of the 17th International Conference on Mining Software Repositories*. 32–42.
[7] Lei Cui, Zhiyu Hao, Yang Jiao, Haiqiang Fei, and Xiaochun Yun. 2021. VulDetector: Detecting Vulnerabilities using Weighted Feature Graph Comparison. *IEEE Transactions on Information Forensics and Security* 16 (2021), 2004–2017.
[8] CVE. 2021. *CVE List*. Retrieved August 31, 2021 from https://cve.mitre.org/cve/
[9] 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*. 1147–1164.
[10] Stanislav Dashevskyi, Achim D Brucker, and Fabio Massacci. 2019. A screening test for disclosed vulnerabilities in foss components. *IEEE Transactions on Software Engineering* 45, 10 (2019), 945–966.
[11] Debian. 2021. *Debian Repository for Advisories*. Retrieved August 31, 2021 from https://salsa.debian.org/security-tracker-team/security-tracker/-/tree/master/data/CVE
[12] Debian. 2021. *Debian Security Tracker*. Retrieved August 31, 2021 from https://security-tracker.debian.org/tracker/
[13] 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*. 869–885.
[14] Ruian Duan, Ashish Bijlani, Yang Ji, Omar Alrawi, Yiyuan Xiong, Moses Ike, Brendan Saltaformaggio, and Wenke Lee. 2019. Automating Patching of Vulnerable Open-Source Software Versions in Application Binaries.. In *Proceedings of the 26th Annual Network and Distributed System Security Symposium*.
[15] Black Duck. 2021. *Black Duck KnowledgeBase*. Retrieved August 31, 2021 from https://www.synopsys.com/content/dam/synopsys/sig-assets/datasheets/bdknowledgebase-ds-ul.pdf
[16] Jiahao Fan, Yi Li, Shaohua Wang, and Tien N Nguyen. 2020. AC/C++ Code Vulnerability Dataset with Code Changes and CVE Summaries. In *Proceedings of the 17th International Conference on Mining Software Repositories*. 508–512.
[17] GitHub. 2021. *GitHub REST API*. Retrieved August 31, 2021 from https://docs.github.com/en/rest/reference/search#search-commits
[18] GitHub. 2021. *GitHub REST API*. Retrieved August 31, 2021 from https://docs.github.com/en/rest/reference/repos#list-branches
[19] GitHub. 2021. *GitHub REST API*. Retrieved August 31, 2021 from https://docs.github.com/en/rest/reference/repos#list-commits
[20] go ethereum. 2021. . Retrieved August 31, 2021 from https://github.com/ethereum/go-ethereum/commit/fb9f7261ec51e38eedb454594fc19f00de1a6834
[21] go ethereum. 2021. . Retrieved August 31, 2021 from https://github.com/ethereum/go-ethereum/commit/83e2761c3a13524bd5d6597ac08994488cf872ef
[22] Red Hat. 2021. *Red Hat Bugzilla*. Retrieved August 31, 2021 from https://bugzilla.redhat.com/
[23] Red Hat. 2021. *WebService API of Red Hat*. Retrieved August 31, 2021 from https://bugzilla.redhat.com/docs/en/html/api/index.html

[24] Jiyong Jang, Abeer Agrawal, and David Brumley. 2012. ReDeBug: finding unpatched code clones in entire os distributions. In *Proceedings of the IEEE Symposium on Security and Privacy*. 48–62.

[25] Jenkins. 2021. . Retrieved August 31, 2021 from https://www.jenkins.io/security/advisory/2019-09-25/#SECURITY-1573

[26] 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 2020 ACM SIGSAC Conference on Computer and Communications Security*. 1149–1163.

[27] Matthieu Jimenez, Yves Le Traon, and Mike Papadakis. 2018. Enabling the continous analysis of security vulnerabilities with vuldata7. In *Proceedings of the IEEE International Working Conference on Source Code Analysis and Manipulation*. 56–61.

[28] Matthieu Jimenez, Renaud Rwemalika, Mike Papadakis, Federica Sarro, Yves Le Traon, and Mark Harman. 2019. The importance of accounting for real-world labelling when predicting software vulnerabilities. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 695–705.

[29] Hyeonseong Jo, Jinwoo Kim, Phillip Porras, Vinod Yegneswaran, and Seungwon Shin. 2021. GapFinder: Finding Inconsistency of Security Information From Unstructured Text. *IEEE Transactions on Information Forensics and Security* 16 (2021), 86–99.

[30] jspwiki. 2021. . Retrieved August 31, 2021 from https://github.com/apache/jspwiki/commit/87c89f0405d6b31fc165358ce5d5bc4536e32a8a

[31] Seulbae Kim, Seunghoon Woo, Heejo Lee, and Hakjoo Oh. 2017. Vuddy: A scalable approach for vulnerable code clone discovery. In *Proceedings of the IEEE Symposium on Security and Privacy*. 595–614.

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

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

[34] Zhen Li, Deqing Zou, Shouhuai Xu, Xinyu Ou, Hai Jin, Sujuan Wang, Zhijun Deng, and Yuyi Zhong. 2018. Vuldeepecker: A deep learning-based system for vulnerability detection. In *Proceedings of the 25th Annual Network and Distributed System Security Symposium*.

[35] Bingchang Liu, Guozhu Meng, Wei Zou, Qi Gong, Feng Li, Min Lin, Dandan Sun, Wei Huo, and Chao Zhang. 2020. A large-scale empirical study on vulnerability distribution within projects and the lessons learned. In *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering*. 1547–1559.

[36] 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*. 919–936.

[37] 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*. 259–268.

[38] Viet Hung Nguyen, Stanislav Dashevskyi, and Fabio Massacci. 2016. An automatic method for assessing the versions affected by a vulnerability. *Empirical Software Engineering* 21, 6 (2016), 2268–2297.

[39] Viet Hung Nguyen and Fabio Massacci. 2013. The (un) reliability of nvd vulnerable versions data: An empirical experiment on google chrome vulnerabilities. In *Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security*. 493–498.

[40] NVD. 2021. . Retrieved August 31, 2021 from https://nvd.nist.gov/vuln/detail/CVE-2019-10077

[41] NVD. 2021. *National Vulnerability Database*. Retrieved August 31, 2021 from https://nvd.nist.gov

[42] NVD. 2021. *NVD Data Feeds*. Retrieved August 31, 2021 from https://nvd.nist.gov/vuln/data-feeds

[43] Ivan Pashchenko, Henrik Plate, Serena Elisa Ponta, Antonino Sabetta, and Fabio Massacci. 2018. Vulnerable open source dependencies: Counting those that matter. In *Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. 1–10.

[44] Ivan Pashchenko, Henrik Plate, Serena Elisa Ponta, Antonino Sabetta, and Fabio Massacci. 2020. Vuln4Real: A Methodology for Counting Actually Vulnerable Dependencies. *IEEE Transactions on Software Engineering* (2020).

[45] Serena Elisa Ponta, Henrik Plate, and Antonino Sabetta. 2020. Detection, assessment and mitigation of vulnerabilities in open source dependencies. *Empirical Software Engineering* 25, 5 (2020), 3175–3215.

[46] Serena Elisa Ponta, Henrik Plate, Antonino Sabetta, Michele Bezzi, and Cédric Dangremont. 2019. A manually-curated dataset of fixes to vulnerabilities of open-source software. In *Proceedings of the IEEE/ACM 16th International Conference on Mining Software Repositories*. 383–387.

[47] ruby saml. 2021. . Retrieved August 31, 2021 from https://github.com/onelogin/ruby-saml/commit/048a544730930f86e46804387a6b6fad50d8176f

[48] ruby saml. 2021. . Retrieved August 31, 2021 from https://github.com/onelogin/ruby-saml/commit/d7ce607d9f9d996e1046dde09b675c3cf0c01280

[49] ruby saml. 2021. . Retrieved August 31, 2021 from https://github.com/onelogin/ruby-saml/commit/a35f7251b86aa3b7caf4a64d8a3451f925e8855c

[50] ruby saml. 2021. . Retrieved August 31, 2021 from https://github.com/onelogin/ruby-saml/commit/03af9e33d2d87f4ac9a644c5b0981ded4dca0bb8

[51] Antonino Sabetta and Michele Bezzi. 2018. A practical approach to the automatic classification of security-relevant commits. In *Proceedings of the IEEE International Conference on Software Maintenance and Evolution*. 579–582.

[52] saltstack. 2021. . Retrieved August 31, 2021 from https://docs.saltstack.com/en/latest/topics/releases/2018.3.3.html

[53] SecurityFocus. 2021. . Retrieved August 31, 2021 from https://www.securityfocus.com

[54] Snyk. 2019. *State of the Open Source Security Report*. Retrieved August 31, 2021 from https://snyk.io/wp-content/uploads/sooss_report_v2.pdf

[55] Snyk. 2021. *Snyk Vulnerability Database*. Retrieved August 31, 2021 from https://snyk.io/vuln

[56] Xin Tan, Yuan Zhang, Chenyuan Mi, Jiajun Cao, Kun Sun, Yifan Lin, and Min Yang. 2021. Locating the Security Patches for Disclosed OSS Vulnerabilities with Vulnerability-Commit Correlation Ranking. In *Proceedings of the 28th ACM Conference on Computer and Communications Security*.

[57] Thrift. 2021. . Retrieved August 31, 2021 from https://issues.apache.org/jira/browse/THRIFT-4647

[58] undertow. 2021. . Retrieved August 31, 2021 from https://issues.redhat.com/browse/UNDERTOW-1430

[59] undertow. 2021. . Retrieved August 31, 2021 from https://github.com/undertow-io/undertow/commit/c46b7b49c5a561731c84a76ee52244369af1af8a

[60] Veracode. 2021. *Veracode Vulnerability Database*. Retrieved August 31, 2021 from https://sca.veracode.com/vulnerability-database/search

[61] Y. Wang, B. Chen, K. Huang, B. Shi, C. Xu, X. Peng, Y. Wu, and Y. Liu. 2020. An Empirical Study of Usages, Updates and Risks of Third-Party Libraries in Java Projects. In *ICSME*. 35–45.

[62] WhiteSource. 2021. *WhiteSource Vulnerability Database*. Retrieved August 31, 2021 from https://www.whitesourcesoftware.com/vulnerability-database/

[63] Yang Xiao, Bihuan Chen, Chendong Yu, Zhengzi Xu, Zimu Yuan, Feng Li, Binghong Liu, Yang Liu, Wei Huo, Wei Zou, et al. 2020. {MVP}: Detecting Vulnerabilities using Patch-Enhanced Vulnerability Signatures. In *Proceedings of the 29th USENIX Security Symposium*. 1165–1182.

[64] 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*. 376–387.

[65] 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 IEEE/ACM 39th International Conference on Software Engineering*. 462–472.

[66] Zhengzi Xu, Yulong Zhang, Longri Zheng, Liangzhao Xia, Chenfu Bao, Zhi Wang, and Yang Liu. 2020. Automatic Hot Patch Generation for Android Kernels. In *Proceedings of the 29th USENIX Security Symposium*. 2397–2414.

[67] Wei You, Peiyuan Zong, Kai Chen, XiaoFeng Wang, Xiaojing Liao, Pan Bian, and Bin Liang. 2017. Semfuzz: Semantics-based automatic generation of proof-of-concept exploits. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 2139–2154.

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

[69] Hang Zhang and Zhiyun Qian. 2018. Precise and accurate patch presence test for binaries. In *Proceedings of the 27th USENIX Security Symposium*. 887–902.

[70] Yaqin Zhou, Shangqing Liu, Jingkai Siow, Xiaoning Du, and Yang Liu. 2019. Devign: Effective Vulnerability Identification by Learning Comprehensive Program Semantics via Graph Neural Networks. In *Proceedings of the 32nd Annual Conference on Neural Information Processing Systems*. 10197–10207.

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