Precise (Un)Affected Version Analysis for Web Vulnerabilities

Youkun Shi*
Fudan University
China
19210240047@fudan.edu.cn

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

Tianhan Luo
Fudan University
China
20210240280@fudan.edu.cn

Xiangyu Mao
Fudan University
China
17307130105@fudan.edu.cn

Min Yang
Fudan University
China
m_yang@fudan.edu.cn

ABSTRACT

Web applications are attractive attack targets given their popularity and large number of vulnerabilities. To mitigate the threat of web vulnerabilities, an important piece of information is their affected versions. However, it is non-trivial to build accurate affected version information because confirming a version as affected or unaffected requires security expertise and huge efforts, while there are usually hundreds of versions to examine. As a result, such information is maintained in a low-quality manner in almost every public vulnerability database. Therefore, it is extremely useful to have a tool that can automatically and precisely examine a large part (even if not all) of the software versions as affected or unaffected.

To this end, this paper proposes a vulnerability-centric approach for precise (un)affected version analysis for web vulnerabilities. The key idea is to extract the vulnerability logic from a patch and directly use the vulnerability logic to check whether a version is (un)affected or not. Compared with existing works, our vulnerability-centric approach helps to tolerate the code changes across different software versions. We construct a high-quality dataset with 34 CVEs and 299 software versions to evaluate our approach. The results show that our approach achieves a precision of 98.15% and a recall of 85.01% in identifying (un)affected versions and significantly outperforms existing tools (e.g., V-SZZ, ReDebug, V0Finder).

CCS CONCEPTS

• **Security and privacy → Web application security**; **Software security engineering**.

KEYWORDS

Web Vulnerability, Vulnerability Database, OSS Security

ACM Reference Format:
Youkun Shi, Yuan Zhang, Tianhan Luo, Xiangyu Mao, and Min Yang. 2022. Precise (Un)Affected Version Analysis for Web Vulnerabilities. In *37th IEEE/ACM International Conference on Automated Software Engineering (ASE '22), October 10–14, 2022, Rochester, MI, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3551349.3556933

*co-first authors

ASE '22, October 10–14, 2022, Rochester, MI, USA

ACM ISBN 978-1-4503-9475-8/22/10…$15.00
https://doi.org/10.1145/3551349.3556933

1 INTRODUCTION

Web applications have a large user base. It is estimated that there are over 1.7 billion websites worldwide, and this number is growing at a rate of 576,000 per day [2]. Meanwhile, security vulnerabilities are also widespread in web applications. According to the testing of Acunetix on 3,500 random-selected websites, more than 90% of web applications have high or medium severity vulnerabilities [4]. As a result, the fast-growing number of web applications attracts a large volume of attacks. For example, a report from PatchStack [21] shows that there are about 30,000 websites compromised every day.

Due to the rapid evolution, web applications often have hundreds of versions. To protect web applications from security vulnerabilities, an essential piece of information is the (un)affected versions of a vulnerability. That is, only with accurate information about the (un)affected versions of a vulnerability, the developers could know which versions to release patches and the website maintainers could know whether their used applications are affected.

Currently, to ease the process of vulnerability reporting and managing, the affected versions of a vulnerability are maintained by several public vulnerability databases [9, 10, 16]. For example, NVD (National Vulnerability Database) [9] is the largest public vulnerability database which indexes all the vulnerabilities in CVE (Common Vulnerabilities and Exposures) [6] and provides the affected versions of a vulnerability in a specific "Known Affected Software Configurations" part of its page. However, since there are a lot of versions for an application, it is extremely difficult to examine all software versions by human experts. Taking NVD as an example, it simply treats all the versions before the reported version as affected [41, 45]. As a result, it is unsurprising to find that the affected version information in almost every vulnerability database has flaws [31, 54, 64, 67]. This calls for a precise (un)affected version analysis tool, which could automatically identify a large part (even if not all) of the software versions as affected or unaffected; thus, at least the efforts in examining these versions could be saved.

An intuitive way to determine whether a version is affected or unaffected is dynamic testing. That is to say, one can run the PoC (Proof of Concept) input of a vulnerability on the version and give a result by observing whether the vulnerability is triggered. However, this method fails for two reasons. First, the PoC inputs are not widely available due to their severe threats to the affected

applications [1, 3]. According to our experience in constructing the dataset in §4.1, more than 60% of the vulnerabilities do not have public PoC inputs. Second, due to the cross-version code changes, the original PoC input sometimes needs to be adjusted a little to trigger the same vulnerability on lower versions [30]. During our dataset construction in §4.1, we need to adjust 14.71% of the collected PoCs to make them work on lower versions. For these reasons, dynamic testing is not an appropriate choice.

Except for dynamic testing, another choice is static analysis. A direct way is to statically analyze the code logic of every software version to verify whether the vulnerability is there or not. However, existing program analysis techniques are still not perfect and meet severe limitations in reasoning complex code, e.g., constraint solving [23, 59, 66, 69], path exploration [50, 68]. To facilitate (un)affected version analysis, we find two lines of work in static analysis that may help. The first line of work is the *code clone analysis* [34, 37, 53, 61, 64, 65] which creates code signatures based on the patch and identifies versions with similar code as affected. The other line of work is the *vulnerability-introducing commit analysis* [27, 33, 38, 41, 57] which uses the patch to locate the introducing commit of the vulnerability and then identifies the versions released between the introducing commit and the patch commit as affected while marking other versions as unaffected. However, we find that both lines of works could not provide precise (un)affected version analysis. As it will be elaborated later in §2.1, the root cause is concluded as the *inappropriate patch assumption* problem, i.e., these works assume that a patch only contains vulnerability-relevant changes and must have deletion lines. This problem ultimately makes existing works hard to accommodate the cross-version code changes, which is a fundamental challenge in (un)affected version analysis.

In light of this, we propose a *vulnerability-centric* approach to enable precise (un)affected version analysis. Similar to existing works, our approach also takes the patch as input; however, it significantly differs in the way of using the patch. Our high-level idea is to first extract the vulnerability logic from the patch and then perform the (un)affected version analysis based on the vulnerability logic itself. The vulnerability-centric design brings advantages two-fold. First, our analysis could be more precise because we directly use the vulnerability logic in the analysis, while existing works do not. Second, our analysis is more resilient to cross-version code changes because vulnerability-irrelevant codes are excluded from the analysis. Moreover, to avoid complicated code reasoning, we adopt a conservative strategy in (un)affected version analysis. To be specific, we only report (un)affected versions with high confidence, even though we may fail to give results for some versions. Nevertheless, according to our evaluation, this strategy helps to make the analysis quite precise and meanwhile helps to identify a large part of versions as (un)affected. Therefore, we believe our approach makes a first and important step towards building accurate affected version information for public vulnerability databases.

Our approach is named AFV (AFfected Versions) and a prototype targeting PHP applications is implemented given the popularity of the PHP language on the web. By constructing a ground-truth dataset with 34 CVEs and 299 software versions (resulting 5,002 vulnerability-version pairs), AFV achieves a precision of 98.15% and a recall of 85.01% in identifying affected and unaffected versions. Compared with start-of-the-art works in vulnerability-introducing commit analysis and code clone analysis (i.e., V-SZZ, ReDebug, V0Finder), AFV significantly outperforms them in terms of both precision and recall. Besides, AFV is very efficient. On average, it costs 97.49 seconds to finish the analysis on one vulnerability-version pair. The evaluation results demonstrate AFV as a useful tool to precisely identify (un)affected versions of a vulnerability.

In summary, we make the following contributions in the paper:

- We propose a vulnerability-centric approach for (un)affected version analysis and design several new techniques for representing, extracting, and using the vulnerability logic for (un)affected version analysis.
- We construct a high-quality and large-scale dataset to measure the effectiveness of (un)affected version analysis, in which the affected versions are all confirmed with PoC inputs and the unaffected versions are all confirmed manually.
- We implement a prototype of our approach and report its results in our dataset, including the comparison results with several state-of-the-art baselines.

The rest of this paper is organized as follows. §2 illustrates the research problem and presents our key idea. §3 describes the overall approach. §4 evaluates the proposed approach and compares with several state-of-the-art works. §5 discusses some issues of the work. §6 summarizes the related work and §7 concludes the paper.

2 MOTIVATION

2.1 Problem Statement

In this paper, we investigate the *(un)affected version analysis* problem for web vulnerabilities. The problem can be formulated as follows. A web application usually has a lot of versions (*V0, V1, .., Vn*). When a vulnerability is discovered on a specific version (i.e., the pre-patch version), it may also affect several other versions. The (un)affected version analysis aims to assess which versions are affected by the vulnerability and which versions are unaffected.

In essence, (un)affected version analysis is quite important. On the one hand, the affected versions are the most fundamental information in mitigating a vulnerability. Hence, we can find such information is included in every vulnerability database, e.g., NVD [9], SecurifyFocus [16], and Openwall [10]. On the other hand, the high importance of vulnerability-affected versions also requires them to be highly accurate. However, prior works show that such information is maintained in a low-quality manner [31, 41, 64]. Existing works also show that it is non-trivial for human experts to examine whether a version is affected [43], due to the complicated vulnerability logic reasoning. Therefore, a precise (un)affected version analysis would significantly help to maintain a high-quality vulnerability database by saving the efforts of security analysts.

A fundamental challenge in (un)affected version analysis is to deal with the code changes across versions. In other words, a precise (un)affected version analysis should keep resilient to the code changes that are irrelevant to the vulnerability. However, we find most of the existing works may fall short. We organize these works into two categories and break down the reasons separately.

❶ *Limitations of Code Clone Analysis.* Vulnerability-affected versions might be identified through code clone analysis, i.e., if

a version of the software contains code whose similarity with the vulnerable code exceeds a pre-defined threshold, it is identified as affected. Among these works, a common way is to use function-level code analysis to search vulnerable code clones [53, 61, 64]. Since the difference between the patched code and the vulnerable code might be minor, recent works use the patch information to optimize the vulnerable code clone analysis and achieve good performance in detecting vulnerable code clones, e.g., ReDebug [34], VUDDY [37] and MVP [65]. However, we find that these works can not support precise (un)affected version analysis, because security patches usually contain vulnerability-irrelevant code changes. For example, according to our analysis in §4.1, 49.60% of patched lines are irrelevant to the vulnerability logic. These vulnerability-irrelevant code changes in a patch not only distract the analysis from the vulnerability logic but also make the analysis hard to adapt to cross-version code changes.

❷ *Limitations of Vulnerability-introducing Commit Analysis.* Vulnerabilities are a kind of bugs. By locating the bug-introducing commit, the versions that are released between the introducing commit and the patch commit can be recognized as affected ones while the others can be recognized as unaffected ones. Following this philosophy, existing works use various ways to locate the bug-introducing commit. Among these works, SZZ [57] and its variants [27, 33, 38, 41] are well-known for their good performance. V-SZZ [41] is a recent work that extends the SZZ algorithm to adapt to the locating of the vulnerability-introducing commits. The basic idea of V-SZZ is to backtrace the commit history to locate the first commit in the code repository that introduces the deletion lines of a security patch. However, we observe two limitations of these works when performing (un)affected version analysis. First, vulnerability-irrelevant patch modifications may bring the noise in the tracing of cross-version code changes, thus degrading the precision of locating vulnerability-introducing commits. Second, the SZZ algorithm can not trace addition lines while according to our analysis in §4.1, security patches may (i.e., 15.90%) contain pure addition lines.

From the above analysis, we conclude that patches are widely used by existing works; however, these works have an *inappropriate assumption* about the patch: 1) the changes in a patch are relevant to the vulnerability, and 2) a patch should have deletions. These assumptions make them hard to tolerate the cross-version code changes and fail to provide precise (un)affected version analysis.

2.2 Our Idea

Different from existing works that directly use the patch in the cross-version code analysis, we propose a *vulnerability-centric* approach. The high-level idea is to extract the vulnerability logic from the patch and use the vulnerability logic for (un)affected version analysis. By keeping the analysis focused on the vulnerability logic itself, our approach could be more resilient to cross-version code changes and enable more precise (un)affected version analysis.

Following the high-level idea, we have two specific questions to explore: ❶ how to represent the vulnerability logic and ❷ how to use the vulnerability logic for (un)affected version identification.

- For the first question, we observe that there is usually a *dangerous function* in a web vulnerability, and a patch fixes the vulnerability by restricting the execution of the dangerous function either in control flow (e.g., preventing attacker-controlled executions) or in data flow (e.g, sanitizing attacker-controlled inputs). Thus, by proposing an impact analysis on the patch, we could locate the dangerous function. Then, the code that determines the behaviors of the dangerous function represents the vulnerability logic.
- For the second question, we want to keep conservative in using the vulnerability logic to identify (un)affected versions. That is to say, we only report (un)affected versions that we are quite sure of. Technically, we only identify a version as affected if it contains a dangerous function with exactly the same vulnerability logic and identifies a version as unaffected only when the dangerous function is absent or patched. Note that we do not aim to identify all the affected and unaffected versions, which require heavy-weight vulnerable logic reasoning. Instead, we want to make the first step to building accurate affected versions of a vulnerability by providing a precise (un)affected version analysis.

2.3 Running Example

We use a real-world example to further illustrate our motivation. Figure 1 shows the patch for CVE-2018-15139, an arbitrary file upload vulnerability reported in OpenEMR 5.1.0.3. The root cause of the vulnerability is that the parameter (`$imagepath`) of `move_uploaded_file()` (line 23) can be controlled by an attacker, leading to the upload of an arbitrary PHP file. The vulnerability is fixed by introducing checks on the extension of the uploaded files (lines 11-14).

Obviously, it is a quite dangerous vulnerability that enables remote code execution. To prevent the vulnerability from being exploited, we first need to know which versions of OpenEMR are affected. When referring to NVD, we find that there are 38 versions (i.e., 2.0.1.2 and 2.7-5.0.1.3) listed in the "Known Affected Software Configuration" part [7]. However, by examining the code of all the versions of OpenEMR, we find that 13 versions (i.e., 2.7.2-3.2.0) that are referred to as affected in NVD are actually unaffected. In fact, these unaffected versions even do not have the source file that contains the vulnerability (i.e., `interface/super/manage_site_files.php`). Such wrong information about affected versions would mislead developers and maintainers when mitigating vulnerabilities and make them disregard the future information provided by the vulnerability database.

We also use this vulnerability to illustrate how existing works and our approach to perform the (un)affected version analysis.

❶ *How existing code clone analysis fails?* V0Finder [64] is a function-level vulnerable code clone detector. It fails to find vulnerable code clones for CVE-2018-15139 because the vulnerable code (line 23) is not in a function. Besides, we find the patch of CVE-2018-15139 contains a lot of vulnerability-irrelevant modifications, such as functionality updates, and code refactoring. To be specific, the patch modifies 17 files, and only 3 lines of the total 488 modification lines are used to fix this vulnerability. As a result, our experiment on ReDebug [34] (a patch-enhanced vulnerable code clone detector) shows that it wrongly identifies 6 unaffected versions as affected by using vulnerability-irrelevant patch modifications to search vulnerable code. MVP [65] is a more recent tool in this line of work,

```
1   $OE_SITE_DIR=dirname(dirname(__FILE__)).'/sites/'.$_SESSION['site_id'];
2   $imagedir = "$OE_SITE_DIR/images";
3   $form_dest_filename = $_POST['form_dest_filename'];
4   if ($form_dest_filename == '') {
5     $form_dest_filename = $_FILES['form_image']['name'];
6   }
7   $form_dest_filename = basename($form_dest_filename);
8   if ($form_dest_filename == '') {
9     die(htmlspecialchars(xl('Cannot find a destination filename')));
10  }
11+ $path_parts = pathinfo($form_dest_filename);
12+ if(in_array(strtolower($path_parts['extension']),
                        array('gif','jpg','jpe','jpeg','png','svg'))) {
13+    die(xl('Only images files are accepted'));
14+ }
15  $imagepath = "$imagedir/$form_dest_filename";
16  if (!is_dir($imagedir)) {
17    mkdir($imagedir);
18  }
19  if (is_file($imagepath)) {
20    unlink($imagepath);
21  }
22  $tmp_name = $_FILES['form_image']['tmp_name'];
23  if(!move_uploaded_file($_FILES['form_image']['tmp_name'],$imagepath)){
24    die(htmlspecialchars(xl('Unable to create') . " '$imagepath'"));
25  }
```

Figure 1: The patch for CVE-2018-15139. Note: the lines with blue number are patch-affected statements; the lines on light yellow background represent vulnerability fingerprint.

but it is not open-source. Though we cannot conduct experiments with MVP, our theoretical analysis shows that it also meets similar problems due to vulnerability-irrelevant patch modifications.

❷ *How existing vulnerability-introducing commit analysis fails?* V-SZZ [41] is recent work on locating the vulnerability-introducing commit. When using V-SZZ [41] to locate the introducing commit for this vulnerability, we find that it cannot locate the correct one. Despite the issues about a large number of vulnerability-irrelevant modifications in this patch, we can observe that this vulnerability is fixed by adding new statements to check the extension of the uploaded files (lines 11-14 in Figure 1). As a result, it is not surprising to find that V-SZZ cannot locate the correct vulnerability-introducing commit with these addition lines. Similar to V-SZZ, other works on identifying bug-introducing commits [27, 33, 38] would also fail due to the same reason.

❸ *How our approach succeeds?* To illustrate our vulnerability-centric approach, here we give an overview of how it works on this example and give more details in §3. First, we locate all the affected statements by the patch (i.e., lines 15-17, 19-20, and 22-24) and find that `move_uploaded_file()` (line 23) is a dangerous function for file upload vulnerability, so it is identified as the dangerous function for this vulnerability. Thereafter, we extract all the code statements that affect the behaviors of this dangerous function from both control dependencies and data dependencies (i.e., lines 1-5, 7-8, 15-16, 19, and 23) and use them to represent the vulnerability logic of this dangerous function. Finally, we use these statements to check whether another version has the same statements for the same dangerous function. If true, this version is affected for having the same vulnerability logic. If this version has no corresponding dangerous function or this version has been patched, it is unaffected. Following this philosophy, we could

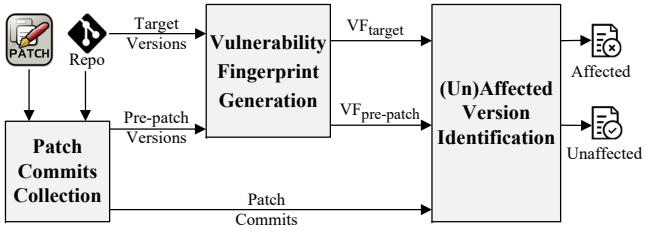

Figure 2: The Architecture of AFV.

precisely identify the (un)affected versions for this vulnerability, including finding its 13 incorrect affected versions in NVD.

3 DESIGN

Following the key idea in §2.2, we present AFV, an automated approach for (un)affected version analysis on web vulnerabilities. To ease the analysis, we first introduce a definition of vulnerability fingerprint (§3.1) and then elaborate on the design of AFV. As shown in Figure 2, AFV consists of three modules.

- *Patch Commits Collection (§3.2)* module takes reference patch commits as input and collects other patch commits on different branches in the code repository. This module helps to collect more patch commits for (un)affected version identification.
- *Vulnerability Fingerprint Generation (§3.3)* module locates the dangerous function in the pre-patch version and generates a fingerprint for this dangerous function. Besides, this module also generates fingerprints for dangerous functions in target versions to support (un)affected version identification.
- *(Un)Affected Version Identification (§3.4)* module identifies affected and unaffected versions with the help of the collected patch commits and the generated vulnerability fingerprints.

3.1 Vulnerability Fingerprint Definition

The high-level idea of AFV is to check the presence of the vulnerability logic on a target version. In the following, we first introduce the concept of *dangerous function* and then use this concept to define *vulnerability fingerprint* which represents the vulnerability logic.

Dangerous Function. Web vulnerabilities usually involve dangerous functions. For example, the dangerous function for a SQL Injection vulnerability might be `mysql_query()`, `pg_query()`, etc. In fact, the root cause of web vulnerabilities is a kind of capability leak of their corresponding dangerous functions. Through these vulnerabilities, an attacker could gain the control of the dangerous functions to achieve some advantages, e.g., controlling `unlink()` to delete files, controlling `move_uploaded_file()` to upload files. Thus, a dangerous function depicts the aim of an attacker in exploiting a vulnerability.

Vulnerability Fingerprint. To check the dangerous function in a target version is affected by a vulnerability or not, the vulnerability fingerprint should accurately represent the code behaviors of the *dangerous function*. Therefore, we define the vulnerability fingerprint as a set of code statements that directly or indirectly affect the execution of the dangerous function. More specifically, we consider two kinds of statements in the vulnerability fingerprint: ❶

the assignment statements that may change the parameters of the dangerous function via data flows, and ❷ the conditional statements that determine the execution of the dangerous function from control flows. Taking Figure 1 as an example, the dangerous function for this vulnerability is `move_uploaded_file()`. Following the data flows and control flows, we could determine the statements on lines 1-5, 7-8, 15-16, 19, and 23 as the vulnerability fingerprint. Though there are 488 lines added/deleted in this patch, we could find only 3 lines is relevant to the vulnerability logic.

3.2 Patch Commits Collection

Patch commits serve as an important input in our approach. They help to understand the vulnerability logic (see §3.3) and identify patched versions (see §3.4). Considering that security patches are usually developed at a code branch (e.g., the master branch) and then backported to other code branches [55], the first step in AFV is to collect the patch commits on other branches to find more kinds of fixes to the same vulnerability.

By studying the practice of software development and patch management, we observe that the backported patch commits and the original patch commits have strong connections in several dimensions. Thus, we devise a searching method to leverage such connections to locate other patch commits in the Git repository. In particular, given some reference patch commits, we collect more patch commits using the following connections:

- *Cherry-picked pattern.* Developers can use "`git cherry-pick`" to apply patches from one branch to another branch. In this case, a sentence will be automatically inserted into the message of the new commit, such as "`cherry picked from commit xxx`" [41].
- *Same code diff.* If a commit on another branch has the same code diff as a patch commit, it can be assumed to be a patch commit for the same vulnerability.
- *Same commit title and message.* We observe that developers often use the same title and message for the patch commits of the same vulnerability on different branches.

Note that prior works also collect patch commits in the code repository [41, 45, 62]. Compared with these works, our approach leverages more ways to facilitate the patch commit collection. §4.5 also evaluates the usefulness of our patch collection module.

3.3 Vulnerability Fingerprint Generation

According to the definition of vulnerability fingerprint in §3.1, AFV first locates the dangerous function of the vulnerability and then extracts all the code statements that directly or indirectly affect the execution of the dangerous function as the vulnerability fingerprint.

Locating the Dangerous Function. Web vulnerabilities are usually caused by the capability leak of their corresponding dangerous functions and are fixed by restricting the executions of these dangerous functions from either control flows or data flows. Thus, we first design a dependency analysis to locate all the code statements that are affected by the patch and then identify the dangerous function from these patch-affected code statements with the help of a pre-defined dangerous function map.

Step 1: Locate Patch-affected Code Statements. We use forward taint analysis to locate the statements affected by a patch. It runs by iteratively propagating the following three types of statements:

- *Assignment statement* may affect the reaching condition/data-flow values of a dangerous function. We taint its assigned variable and locate all the statements that use this variable with a program dependency analysis. They are all used for the taint analysis.
- *Conditional statement* may change the reaching condition of a dangerous function, or its subsequent statements may affect a dangerous function. We include all its subsequent statements for the next round of taint analysis.
- *Exit statement* (e.g., `die()`, `exit`) is similar to a conditional statement, which affects the execution of all subsequent statements. Thus, we also use its subsequent statements for the taint analysis.

Step 2: Identify Dangerous Function. According to the root cause of web vulnerabilities, we observe some common dangerous functions for each type of web vulnerability. Thus, we could identify the dangerous function from the patch-affected code statements with the help of a pre-defined function map (see Table 1). Take Figure 1 as an example, we identify line 15-17, 19-20 and 22-24 as patch-affected statements. Since it is an arbitrary file upload vulnerability, we can recognize line 23 (`move_uploaded_file()`) as the dangerous function. Note that some dangerous functions may be customized by developers. To handle these cases, we perform inter-procedure analysis to locate the final dangerous functions. More specifically, for the function calls in the patch-affected statements, AFV will follow the call edges to further analyze the patch-affected code statements in the callee functions. This step is executed iteratively (with a threshold, e.g., 5 in our setting), until AFV locates the final dangerous function in the patch-affected statements.

Generating Vulnerability Fingerprint. The vulnerability fingerprint is a set of code statements that directly or indirectly affect the execution of the dangerous function. We combine several static analysis techniques to extract these code statements.

- First, AFV performs backward taint analysis on the parameters of the dangerous function to trace the source variables that affect the values of the parameters.
- Second, AFV collects all the code statements that may be executed between the source variables and the dangerous function with forward control-flow analysis. Among these statements, AFV checks each assignment statement and conditional statement to determine whether it would affect the execution of the dangerous function through program dependency analysis. All such statements are extracted for further analysis.
- Finally, AFV normalizes the extracted statements by removing all non-ASCII characters and whitespace characters (e.g., $\backslash t$, $\backslash n$) and converting all characters into the lower case. These statements comprise the fingerprint of the dangerous function.

3.4 (Un)Affected Version Identification

AFV generates vulnerability fingerprints from all the patches. Based on the collected patch commits and the generated vulnerability fingerprints, AFV identifies affected and unaffected versions. With the aim to aid the maintaining of the affected version information of a vulnerability database, AFV adopts a conservative strategy

Table 1: Dangerous Functions for Some Vulnerability Types.

Vulnerability Type	Dangerous Function
Server-Side XSS	echo, print, print_r
SQL Injection	pg_query, pg_send_query, pg_prepare, mysql_query, mysqli_prepare, mysqli_query, mysqli_real_query
Arbitrary File Manipulation	include, include_once, require, require_once, file_put_contents, fopen, fwrite, file, readfile, unlink, rmdir
Command Injection	exec, passthru, proc_open, system, shell_exec, popen, pcntl_exec
Code Injection	eval, create_function, assert, array_map
Executable File Upload	copy, fopen, move_uploaded_file, rename
Open Redirect	header
PHP Object Injection	unserialize

in identifying (un)affected versions. More specifically, AFV favors precision than recall because we expect the versions identified by AFV (either affected or unaffected ones) do not require further manual efforts in confirmation. Technically, there are three steps in identifying the (un)affected versions:

- First, AFV checks whether the target versions are patched by analyzing the commit history of the code repository. If the version is the parent of any patch commit, it is patched and deemed as *unaffected*.
- Second, AFV locates the dangerous function in the target version. If no corresponding dangerous function is located, this version is identified as *unaffected*.
- Third, AFV generates a fingerprint of the dangerous function (VF_{target}) on the target version and compares it with the vulnerability fingerprint ($VF_{pre-patch}$). If VF_{target} is exactly the same with $VF_{pre-patch}$, this version is deemed as *affected*.

Note that some versions may still not be labeled after the above three steps; they are deemed as *unknown*. The benefits of AFV reflect in helping human experts to avoid examining a large part of versions (aka those identified versions).

4 EVALUATION

4.1 Experimental Setup

Prototype. We implemented a prototype of AFV targeting PHP applications, which consists of 4,268 lines of Python code. The *Patched Commit Collection* module is built upon the `GitPython` library; the *Vulnerability Fingerprint Generation* and *(Un)Affected Version Identification* modules are based on the PHPJoern static analysis framework [26]. Besides, we also enhanced PHPJoern to support inter-procedure analysis and virtual call resolving with a class hierarchy analysis and optimized its Code Property Graph construction by supporting several new PHP syntaxes, e.g., `switch` statements. By thoroughly examining the PHP manual [12] and other resources [5, 8, 11], we collected a representative list of dangerous PHP functions for common vulnerability types in Table 1. Note that modelling dangerous functions in PHP program analysis tasks [25, 26, 47, 63] is quite common.

Experiments. Our evaluation is organized by answering the following research questions:

- RQ1: How effective is AFV in identifying the affected and unaffected versions for a given vulnerability?
- RQ2: How effective is AFV when compared with some related approaches?
- RQ3: How efficient is AFV in performing the analysis?
- RQ4: How does the patch commit collection module perform?

Dataset. Our evaluation requires a vulnerability dataset with labeled (un)affected status on various versions of the related software. Since there is no public dataset of this kind for PHP web applications, we have to construct a new dataset. The most challenging part in constructing such a dataset is the labeling of the (un)affected status of a vulnerability on various software versions. Meanwhile, the quality of such information is very important to measuring a tool like AFV. To this end, we have two requirements to guarantee the quality of labeling the (un)affected information: 1) to label an affected version, we require a PoC to trigger the vulnerability; and 2) to label an unaffected version, we require to manually confirm that the dangerous function is absent, patched or can not be triggered.

In addition to the two requirements, the dataset should be representative, and the manual efforts in constructing the dataset should be reduced as much as possible. Therefore, we further set four guidelines in dataset construction: 1) the collected vulnerabilities should cover various common types of web application vulnerabilities; 2) the collected applications should have a certain number of versions (e.g., more than 100 versions) and popular ones are preferred (e.g., more than 1,000 GitHub stars); 3) there should be public PoC inputs and patches for the collected vulnerabilities to aid the (un)affected version labeling; and 4) we should keep the number of applications as small as possible because we have to set up plenty of versions of them for PoC testing.

Following the above guidelines, we collected 26 CVEs from MantisBT[1] and 8 CVEs from Piwigo[2] to construct the vulnerability dataset. The collected CVEs have covered all the vulnerability types in Table 1. Furthermore, for all the 34 CVEs, the developers have published detailed vulnerability information, including the PoC inputs and the patches, which greatly eases the understanding of the vulnerability logic and the labeling of the (un)affected versions. Then, we wrote a crawler to automatically collect 146 versions of MantisBT and 161 versions of Piwigo from GitHub and used them as the application dataset. To guarantee the quality of labeling the (un)affected versions of the vulnerability dataset, three authors have been involved, each of whom has at least 4 years of expertise in web security. The whole process costs over 650 human hours. To be specific, the analysts follow the following process to label the ground-truth dataset:

- First, we manually set up each version of the application in a docker container with the guidance of the official documentation. In all, we successfully set up 145 versions of MantisBT and 154 versions of Piwigo, while there are 8 versions that failed to be built because their dependent MySQL version is too old to be

[1]MantisBT is a popular bug tracking system and has more than 1,400 stars in GitHub.
[2]Piwigo is a photo management system with more than 1,700 stars in GitHub.

supported or their released source code miss some important files (e.g., the installation files). This step costs us about 100 human hours.

- Second, for each vulnerability, two analysts independently analyzed its vulnerability logic (e.g., the dangerous function, the triggering condition, and the path of the vulnerability) with the help of the PoC input and Xdebug [22]. Based on the vulnerability logic, the analysts determined the hosting files for the vulnerability (i.e., the files hosting the dangerous functions). When the two analysts have different opinions, a third analyst would be involved to reach a consensus. After that, we wrote a script to automatically identify the versions that do not have the hosting files of the vulnerability, among all the 5,002 (i.e., 145*26+154*8) vulnerability-version pairs. In all, 860 such versions have been identified and are deemed as *unaffected*. This step takes about 150 human hours.
- Third, we tried to identify the versions that applied the patches. To be specific, two analysts independently analyzed the given security patch(es) to identify the patch-affected files and the patching logic and then tried to locate the patch commits for each application version by tracing the commit history of these files in GitHub and checking the commit code modifications. If the two analysts have located different patch commits for a version, a third analyst would participate. Some scripts have been developed to facilitate the process. In this way, 2,179 patched versions have been identified from the left set (4,142) and are deemed them as *unaffected*. This step takes about 70 human hours.
- Forth, PoC testing is used on the remaining 1,963 versions to observe whether the vulnerability is triggered at the dangerous function. Note that for those versions where the original PoC cannot trigger the vulnerability, the analysts tried to adjust the PoC unless they could confirm that the vulnerability logic does not exist or cannot be triggered. In particular, 1,405 versions have been verified as *affected* with the original PoC, 78 versions have been verified as *affected* with a modified PoC and the remaining 480 versions are verified as *unaffected*. Since PoC testing and adjustment is quite time-consuming, this step costs about 330 human hours.

In all, the ground-truth dataset covers 34 CVEs and 299 software versions, resulting 5,002 vulnerability-version pairs. Among these pairs, 1,483 cases are labeled as *affected* and the left 3,519 cases are labeled as *unaffected*.

For the 34 CVEs, we manually collected 44 patches[3] in the NVD. Among all these patches, we find 7 patches (15.9%) that contain pure insertions. In all, these patches have 1,277 addition lines and 489 deletion lines, covering the modification to 80 files. Based on the analysis of *Vulnerability Fingerprint Generation* module, we find 876 lines (49.60%) are irrelevant to the vulnerability fixing. These statistics confirm our argument about the "inappropriate patch assumption" problem that is met by existing works.

Baselines. We compare AFV with the following two lines of works.

[3]Some CVEs have multiple patch commits in the NVD for two reasons: i) the vulnerability is fixed by multiple commits; ii) the vulnerability is fixed by separated commits on different branches. For the former case, the multiple commits are merged and viewed as a single patch in our evaluation.

Table 2: Effectiveness Results of AFV. (RQ1)

Ground Truth		Tool's Result				
		TP	FP[1]	FN[2]	Precision	Recall
Affected	1,483	1,034	35	449	96.73%	69.72%
Unaffected	3,519	3,218	45	301	98.62%	91.45%
All	5,002	4,252	80	750	98.15%	85.01%

[1]The false positives in AFV represent the versions that have been given results (either affected or unaffected), but the results are wrong.
[2]The false negatives in AFV consist of the versions that have been incorrectly identified as affected or unaffected, and the versions that have not been successfully identified (i.e., identified as unknown).

- *Vulnerability-introducing Commit Analysis.* As described in §2.1, this line of research assesses vulnerability-(un)affected versions by locating the vulnerability-introducing commit. In this line of work, V-SZZ [41] is the state-of-the-art which improves the original SZZ algorithm [57] and outperforms the original SZZ including its variants (e.g., B-SZZ [33], AG-SZZ [38], MA-SZZ [27]). Besides, the source code of V-SZZ is publicly available [18]. Thus, V-SZZ is selected to compare with our tool.
- *Code Clone Analysis.* V0Finder [64] is a recent function-level code clone detector that is used to discover the first version where a vulnerability is introduced. Its source code has been released [19], so it is used as a baseline. Besides, existing works also use patch information to optimize vulnerable code clone analysis, e.g., MVP [65], VUDDY [37] and ReDebug [34]. In this direction, MVP is a recent work; however, it is not open-source. VUDDY only supports the C language. Fortunately, ReDebug is open-source [14] and language-independent, so it is included in our comparison experiment. As analyzed in §2.1, MVP and VUDDY also meet the same limitations as ReDebug in (un)affected versions analysis.

Environment. The experiments are run on a Ubuntu 18.04 machine with an Intel Xeon Gold 6242 processor and 245 GB memory. All the baselines are configured with the same setting as their papers.

4.2 Effectiveness (RQ1)

In this experiment, we use the whole ground-truth dataset to evaluate the effectiveness of AFV in identifying affected and unaffected versions. The detailed results are shown in Table 2. In all, AFV reports 1,069 affected versions and 3,263 unaffected versions while 1,034 and 3,218 of them are true positives. That means the affected version analysis and the unaffected version analysis achieves a precision of 96.73% and 98.62%, respectively. When combining the results on affected and unaffected version identification, the overall precision is 98.15%. Besides, the overall recall is also high, i.e., 85.01%. The evaluation results demonstrate that AFV can significantly save the manual efforts of examining a large part of software versions. Next, we break down the detailed reasons for the false positives and false negatives.

False Positive Analysis. Though AFV adopts a conservative strategy in identifying affected versions (i.e., finding exactly the same vulnerability fingerprint in the target version), it still reports 35 false positives in affected version analysis. By analyzing all

these FPs, we find that they are caused by the same reason: *the entry for the vulnerability-triggering request does not exist, but the vulnerability logic remains the same.* More specifically, we observe that web applications usually use routing rules/configurations for request dispatching, while in these cases there is no dispatching for the vulnerability-triggering requests though the dangerous functions and the vulnerable logic are the same as the pre-patch versions. Since AFV has not taken such rules or configurations into the analysis scope, these versions are marked as unaffected in the ground truth but are identified as affected by AFV, leading to false positives. One possible way for AFV to eliminate such FPs is to take a PoC as input and learn the complete vulnerability triggering condition from the PoC [55, 56].

To guarantee the precision in identifying unaffected versions, AFV only reports a version as unaffected when the version is patched, or there is no corresponding dangerous function in this version. However, from Table 2, we find that AFV still reports 45 false positives in the unaffected version analysis. We have analyzed all these cases and find they are all caused by one reason: *the dangerous functions of these vulnerabilities have been changed to other functions with similar functionalities.* For example, the dangerous function for CVE-2011-3578 in MantisBT 1.2.7 is `echo()`, but in MantisBT 1.2.0a2, it is changed to `print()`. Thus, these versions are indeed affected but are wrongly identified as unaffected by AFV.

False Negative Analysis. There are 750 false negatives that have not been successfully or correctly identified as affected or unaffected by AFV. In these versions, 80 versions are incorrectly identified as affected or unaffected by AFV (aka the 80 FPs which have been analyzed above), and the left 670 versions are identified as unknown by AFV (see the descriptions in §3.4).

Among these 670 versions, 404 versions are true affected versions and the other 266 versions are true unaffected versions. For the 266 unaffected versions that have been identified as unknown by AFV, we find that each of them has the corresponding dangerous function whose fingerprint is not exactly the same with $VF_{pre-patch}$. Identifying these dangerous functions as unaffected requires complicated reasoning about their code logic [23, 59, 66, 69]. For the 404 affected versions that have been identified as unknown by AFV, we find there are two situations:

- *Semantically-equivalent vulnerability fingerprint (299 versions).* For a version in this category, though its vulnerability fingerprint is not exactly the same with $VF_{pre-patch}$, they are equivalent in semantics. For example, the `explode()` function in a fingerprint is changed to `split()` in another fingerprint. To identify these vulnerability fingerprints as the same, some more advanced code equivalence testing techniques [42, 44, 51] would help.
- *Different vulnerability fingerprints (105 versions).* The affected versions in this category have different vulnerability fingerprints from those of the pre-patched versions, but they can be triggered with the same PoC input. Identifying these versions as affected also requires complicated reasoning about the code logic of their dangerous functions [23, 59, 66, 69].

4.3 Comparison (RQ2)

We compare AFV with different groups of baselines.

Table 3: Comparison Results with V-SZZ. (RQ2)

Ground Truth		Tools	Tool's Results				
			TP	FP	FN	Precision	Recall
Affected	1,483	AFV	1,034	35	449	96.73%	69.72%
		V-SZZ	962	1,191	521	44.68%	64.87%
		V-SZZ++[1]	962	585	521	62.18%	64.87%
Unaffected	3,519	AFV	3,218	45	301	98.62%	91.45%
		V-SZZ	2,328	521	1,191	81.71%	66.16%
		V-SZZ++	2,934	521	585	84.92%	83.38%
All	5,002	AFV	4,252	80	750	98.15%	85.01%
		V-SZZ	3,290	1,712	1,712	65.77%	65.77%
		V-SZZ++	3,896	1,106	1,106	77.89%	77.89%

[1]V-SZZ++ is an enhanced version of V-SZZ and will be used in RQ4 (§4.5).

4.3.1 Comparison with V-SZZ. Similar to AFV, V-SZZ can identify both affected and unaffected versions. Thus, we run the comparison experiments on the whole ground truth.

V-SZZ Setup. V-SZZ has provided detailed documentation in the `README.md` [18]. Following these guidelines, we run V-SZZ within three steps. First, we use the official patch commit as input and execute `"python main.py"`, so that V-SZZ can collect the vulnerability-introducing commits and the patch commits in the code repository. Second, based on the collected commits, we run `"python extract_tag.py"` to make V-SZZ identify the affected versions. Third, according to the description in the V-SZZ paper [41], all the remaining versions are marked as unaffected.

Results Overview. The comparison results between V-SZZ and AFV are presented in Table 3. Note that V-SZZ identifies affected versions first and then labels all the remaining versions as unaffected, so its FPs/FNs in affected version analysis correspond to the FNs/FPs in unaffected version analysis, respectively. From this table, we can find that AFV achieves superior performance than V-SZZ in both the precision and the recall. To be specific, AFV outperforms V-SZZ by 49.22% in precision and 29.25% in recall. This clearly demonstrates the advantages of AFV in considering the vulnerability logic other than the patch modifications in performing vulnerability affection analysis. Furthermore, it is surprising to find that the performance of V-SZZ is quite worse than that reported in its paper. We rigorously investigate the reasons by various means, such as studying the paper, examining the source code, and communicating our findings with the authors of V-SZZ. Eventually, we conclude several major reasons that cause V-SZZ to have many false positives and false negatives in our evaluation. We present these reasons in the following.

FPs in Affected Version Analysis. In general, we find two major limitations in V-SZZ that lead to false positives/negatives in identifying affected/unaffected versions.

- *Vulnerability-irrelevant modifications in the patch (282 FPs).* V-SZZ locates the vulnerability-introducing commit by backtracing the patch modifications, so vulnerability-irrelevant modifications in a patch may cause V-SZZ to locate a vulnerability-introducing commit earlier than the real one. In this scenario, V-SZZ would report false positives in identifying affected versions. We observe 282 FPs for this reason.

- *Fail to collect some backported patch commits (909 FPs).* V-SZZ identifies the versions released between the vulnerability-introducing commit and the patch commit as affected and marks the others as unaffected. Hence, if it fails to identify some patch commits, it would wrongly identify some patched versions as affected. We observe 909 FPs due to this reason.

FNs in Affected Version Analysis. We summarize two reasons that cause V-SZZ to report false negatives/positives in identifying affected/unaffected versions.

- *Cannot handle patches with pure addition lines (413 FNs).* V-SZZ uses the deletion lines in a patch to trace the origin commits that introduce these lines and identifies the vulnerability-introducing commit from these origin commits. However, we find 7 patches only contain addition lines. It makes V-SZZ fail to locate the vulnerability-introducing commits for them, causing 413 false negatives in identifying affected versions. In §2.2, we conclude this problem as the "inappropriate patch assumption". In contrast, AFV does not have such assumptions.
- *Identify wrong vulnerability-introducing commits (108 FNs).* During the tracing of the introducing commit of a deletion line, V-SZZ relies on line-level code similarity to determine whether an addition line and a deletion line belong to the same line. However, sometimes V-SZZ may fail to identify them as the same line due to the significant code changes between them, thus breaking the tracing of the deletion line. In this situation, V-SZZ would locate a vulnerability-introducing commit later than the real one, causing false negatives in identifying affected versions. We observe 108 FNs for this reason.

4.3.2 Comparison with ReDebug & V0Finder. Different from AFV, ReDebug and V0Finder can only identify vulnerability-affected versions. Thus, we compare AFV with them in identifying the affected versions of the ground truth.

ReDebug Setup. We run ReDebug with two steps. First, we get a patch diff file from GitHub. Second, we use ReDebug to test whether a software version is affected using this command: `"python redebug.py <patch> <target>"`.

V0Finder Setup. There are three steps. First, we use the patch commit(s) as input and run `"python3 CVEPatch_Collector.py"` for V0Finder to generate vulnerable function signatures. Second, we run `"python3 OSS_Collector.py"` for V0Finder to generate signatures for all the functions in our application dataset. Third, we use the command `"python3 DetectingVulClones.py"` to make V0Finder identify vulnerability-affected versions. Note that *Universal Ctags* [17] is used by V0Finder to extract some function-level information for signature generation, while it currently does not support PHP files. Therefore, we extend V0Finder to support the parsing of PHP files with the help of the *PHP-Parser* [13] library.

Results Overview. Table 4 shows the comparison results with ReDebug and V0Finder. Similarly, we observe that AFV significantly outperforms both tools. Compared with ReDebug, AFV significantly improves both the precision and recall in identifying affected versions. Though V0Finder achieves a comparable precision with AFV, its recall is quite low. To understand the reasons for the bad performance of these two tools, we manually analyze their false positives and false negatives. The causes are summarized below.

Table 4: Comparison Results with ReDebug and V0Finder in Identifying Affected Versions. (RQ2)

Tools	TP	FP	FN	Precision	Recall
AFV	1,034	35	449	96.73%	69.72%
ReDebug	871	146	612	85.64%	58.73%
V0Finder	239	20	1,244	92.28%	16.12%

FPs and FNs in ReDebug. There are three reasons for the inaccuracies of ReDebug.

- *The vulnerability logic exists but can not be triggered (9 FPs).* Similar to the FPs in AFV, ReDebug reports 9 FP versions where the logic of the dangerous function is vulnerable but can not be triggered due to the absence of the vulnerability entry.
- *Vulnerability-irrelevant modifications in the patch (137 FPs).* Due to the lack of understanding of the vulnerability logic, ReDebug might use vulnerability-irrelevant modifications in the patch to identify similar vulnerable codes. In all, it causes 137 false positives. In contrast, AFV does not have such FPs thanks to the *Vulnerability Fingerprint Generation* module, which excludes vulnerability-irrelevant code from the analysis scope.
- *Patch-centric vulnerable code searching (612 FNs).* ReDebug identifies vulnerable code clones using the code lines nearby the patch. However, these nearby lines do not represent the vulnerability logic. Thus, when there are significant code changes, ReDebug might fail to identify the vulnerability logic of an affected version. We find ReDebug fails to identify 612 affected versions due to this reason. In contrast, AFV extracts the logic of a vulnerability into a fingerprint and directly uses the fingerprint to identify affected versions, avoiding a lot of false negatives.

FPs and FNs in V0Finder. Compared with ReDebug, V0Finder also has similar reasons for the false positives. First, it also reports false positives for the 9 versions where the dangerous function is vulnerable but can not be triggered due to the absent entry (*9 FPs*). Second, due to vulnerability-irrelevant modifications in a patch, V0Finder wrongly identifies vulnerable functions. Using these functions in the subsequent affected version analysis reports incorrect affected versions (*11 FPs*). On the other side, we find that the major reason for the false negatives in V0Finder is that the patches for 21 CVEs do not modify a function (i.e., the code modifications occur outside the functions). Thus, these patches are beyond the scope of V0Finder (causing *982 FNs*). For the remaining *262 FNs*, they are caused by including vulnerability-irrelevant code in the function-level code similarity analysis. It makes V0Finder fail to identify affected versions when significant code changes occur. Compared with V0Finder, AFV does not report so many false negatives due to its vulnerability-centric design, which does not rely on function-level code analysis and does not consider vulnerability-irrelevant code in the analysis.

Table 5: Efficiency Results of AFV (in seconds). (RQ3)

AFV Module	Average	Medium
Patch Commits Collection	287.27	249.59
Vulnerability Fingerprint Generation	13,424.94	2,355.00
Vulnerability (Un)Affection Analsysis	631.22	488.15
Total	14,343.43	3,313.27

4.4 Efficiency (RQ3)

We evaluate the efficiency of AFV on the whole dataset. Table 5 presents the results. On average, identifying (un)affected versions of a vulnerability requires to analyze 147.12 (i.e., 5,002/34) software versions and costs 14,343.43 seconds. That is, AFV averagely costs 97.49 seconds to analyze a vulnerability-version pair. Since program analysis is used by AFV to generate *vulnerability fingerprints*, it spends more time to finish the analysis than ReDebug, V0Finder, and V-SZZ. However, though these tools are lightweight, their analysis is quite imprecise. Considering that AFV is mostly used as an offline tool, we believe precision is more favored than efficiency and its efficiency is quite acceptable.

4.5 Patch Commits Collection (RQ4)

Both AFV and V-SZZ collect patch commits in the code repository before identifying affected/unaffected versions. As described in §3.2, AFV uses more ways than V-SZZ in patch commits collection. In this experiment, we report the results of their collected patch commits and the impact on the final analysis results.

Given the manually-collected 44 patches, V-SZZ further locates 17 new patch commits by using the cherry-picked pattern and the code diff information. In addition to the ways used by V-SZZ, AFV also uses the commit title and message information, which helps to locate the backported patch commits that are not created using `"git cherry-pick"` nor `"git merge"`. In all, AFV locates 25 new patch commits. It helps AFV to identify 2,179 patched versions while V-SZZ can only identify 1,573 patched versions. Note that all the new patches collected by AFV and V-SZZ are found to be correct.

To measure the impact of patch commits collection on the final analysis results, we incorporate our patch commits collection technique into V-SZZ and name the enhanced variant as V-SZZ++. We also compare AFV with V-SZZ++. The results are presented in Table 3. We find that our patch commits collection technique significantly improves the accuracy of V-SZZ++, while V-SZZ++ still reports more incorrect affected/unaffected versions than AFV.

5 DISCUSSION

Threats to Validity. ❶ The validity of the constructed dataset in the evaluation might bring threats to the results. As described in §4.1, we have devised several mechanisms to mitigate the constructed threat. First, we set two requirements to mitigate the threat of labeling the (un)affected versions: (1) all the affected versions are verified with PoCs, and (2) all the unaffected versions are manually confirmed. Besides, our labeling process has involved three security analysts. They worked together to ensure the quality of the ground truth. As described in §4.1, before the labeling, all the analysts studied the root cause of every vulnerability in the dataset including its patch semantics with the help of the PoC inputs and Xdebug. Meanwhile, every labeled version has been checked by two analysts, and a third analyst would be involved if they cannot reach an agreement. Second, we set four guidelines for selecting the web applications and their vulnerabilities into the dataset. All the selected applications are quite popular, and all the vulnerabilities are selected from public vulnerability databases. Besides, all the versions of the selected applications in GitHub are included in the dataset.

❷ The scale of the dataset might also bring threats to validity. In all, the dataset consists of 5,002 vulnerability-version pairs, covering 34 vulnerabilities and 299 versions. Among all the vulnerability-version pairs, there are 1,483 affected cases and 3,519 unaffected cases. We think the scale of the dataset is representative.

❸ The adoption of existing tools might bring threats to the comparison results. As described in §4.3, both V-SZZ and ReDebug can be directly applied to PHP code and we have followed their guidelines [14, 18] in the comparison experiment. For V0Finder, it uses *Universal Ctags* [17] to extract function-level information for signature generation, but the tool does not support PHP code. Thus, we leverage the *PHP-Parser* [13] library to make V0Finder support parsing PHP files. Note that V0Finder just needs the function name and its corresponding start and end line numbers to collect function-level information, while such information can be directly acquired by *PHP-Parser* with built-in APIs. Therefore, we believe our adaption of V0Finder does not hurt its original performance.

Trade-off between Precision and Recall. Accurate affected versions of a vulnerability are an important piece of information in vulnerability management. However, due to the diverse code changes across versions, building affected versions of a vulnerability usually meets a trade-off between precision and recall. Previous work [45] chooses to report as many affected versions as possible, which sacrifices precision for recall. However, the imprecision of such information may not only waste the time of webmasters in fixing unaffected versions of a vulnerability but also make them disregard the future provided information. Different from previous works, we adopt a different philosophy. First, we not only identify affected versions but also identify unaffected versions. Second, we only report affected/unaffected versions with high confidence, while keeping the left versions as unknown ones. The advantage of such design is that it brings a clear boundary between the reliable labeled versions and those that need further investigation. In short, our conservative design trades recall for precision. Fortunately, our proposed approach also achieves a high recall of 85.01%.

Further Improvement. The evaluation results in §4.2 show that AFV achieves a high precision (98.15%) and recall (85.01%). The precision of AFV can be further improved. First, the FPs in identifying affected versions can be mitigated with the help of the PoC analysis to understand the complete vulnerability triggering condition. Second, the FPs in identifying unaffected versions can be eliminated by identifying those dangerous functions with similar functionalities as the same. With the aim to be precise, AFV reports more FNs than FPs. According to the analysis in §4.2, avoiding these FNs require heavyweight program analysis techniques to reason the code logic of the dangerous functions and perform semantic-level code equivalence testing. The contribution of our work is to

make the first step in providing a lightweight and precise analysis to identify a large part of (un)affected versions while leaving the rest of versions as the future work. Considering that there are a lot of versions to examine, we believe this step is important and necessary.

Generalization. Though the prototype of AFV is implemented for PHP web applications, the overall approach is generally applicable to web applications of other languages. First, the *Patch Commits Collection* module is based on code commit search; thus it is language-independent. Second, to support a new programming language, the *Vulnerability Fingerprint Generation* module and the *(Un)Affected Version Identification* module could be implemented with another static analysis framework for the new language. For example, to make AFV support Java web applications, Soot [60] or Wala [20] can be used to implement these two modules. That said, adapting these two modules for a new programming language requires considerable engineering efforts. To be specific, one should re-implement several analysis tasks on top of the new static analysis framework, e.g., forward taint analysis to locate the dangerous function, and backward taint analysis to trace the source variables that affect the dangerous function. Besides, as a vulnerability-centric approach for (un)affection version analysis, the whole analysis of AFV is built upon the dangerous function of a vulnerability. Therefore, our approach supports various application types and vulnerability types, but it does not support applications/vulnerabilities that do not have dangerous functions.

6 RELATED WORK

(Un)Affected Version Analysis. The information about the affected versions of a vulnerability is quite important. To this end, existing works have explored both static analysis [41] and dynamic testing [30] techniques. Dai *et al.* [30] proposed a PoC migration approach that takes a PoC as input and migrates the PoC to verify other affected versions. However, according to our experience of collecting the PoC inputs for web vulnerabilities in §4.1, more than 60% of the vulnerabilities do not have public PoC inputs. Therefore, this approach is inappropriate for web vulnerabilities. V-SZZ [41] takes a patch as input and backtraces the deletion lines of the patch across the code commit history to locate the vulnerability-introducing commit. Based on the vulnerability-introducing commit, all the versions released between it and the patch commit can be identified as affected. However, V-SZZ cannot locate the vulnerability-introducing commit for a patch with pure addition lines and its performance degrades significantly when there are vulnerability-irrelevant modifications in the patch. Compared with V-SZZ, our vulnerability-centric approach does not have such limitations and achieves superior performance.

Vulnerable Code Clone Detection. Many approaches have been proposed to detect vulnerable code clones [34, 37, 39, 40, 65]. Woo *et al.* [64] attempted to locate the origin of a vulnerability by introducing a function-level vulnerable code clone analysis. Jang *et al.* [34] proposed ReDeBug to search unpatched code clones in the code base of an operating system. Xiao *et al.* [65] generated patch-enhanced vulnerability signatures to accurately detect vulnerable and unpatched code clones.

It can be found that these works commonly use patches to identify vulnerable code and search for vulnerable code clones. To some extent, AFV is similar to these works for also using patches to analyze the vulnerability. However, AFV significantly differs from existing works in the way of using the patch. Technically, our approach tries to understand the logic of a vulnerability from its patch by locating its dangerous function and extracting the vulnerability logic. This design helps AFV to exclude the vulnerability-irrelevant patch modifications from the analysis scope, thus being more resilient to cross-version code changes than existing works.

Vulnerability Detection. (Un)Affected version analysis of a vulnerability is used after the detection of the vulnerability. There is a lot of work in detecting web vulnerabilities. A commonly used technique is static analysis [26, 28, 29, 35, 46, 58], but it bears high false positives due to the lack of PoCs. As a dynamic analysis technique, fuzzing [32, 48, 49, 52] has become quite popular recently. Its advantage is that it can generate PoCs, and its disadvantage is the code coverage. Compared with static and dynamic analysis, hybrid analysis [24, 25, 36] might be a more preferable approach by combining both advantages of them. However, none of these works can be directly used to determine whether a version is affected or unaffected by a specific vulnerability.

7 CONCLUSION

This paper proposes AFV, a vulnerability-centric approach for precise (un)affected version analysis for web vulnerabilities. The high-level idea is to understand the vulnerability logic with the help of the patch and use the vulnerability logic to check whether a version is (un)affected or not. Since only the vulnerability logic is used in the analysis, AFV is more resilient to cross-version code changes than existing works, which is a fundamental challenge in (un)affected version analysis. To conduct a representative evaluation, a high-quality large-scale dataset is constructed, in which the affected versions are all confirmed with PoC inputs, and the unaffected versions are all confirmed manually. The results demonstrate AFV as a useful tool in automatically and precisely examining a large part of the software versions as (un)affected. The dataset and the source code are publicly available [15].

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their insightful comments. This work was supported in part by the National Natural Science Foundation of China (U1836210, U1836213, 62172105, 61972099, 62172104, 62102091, 62102093), the National Key R&D Program of China (2021YFB3101200), and the Natural Science Foundation of Shanghai (19ZR1404800). Yuan Zhang was supported in part by the Shanghai Rising-Star Program 21QA1400700 and the Shanghai Pilot Program for Basic Research-Fudan University 21TQ1400100 (21TQ012). Min Yang is the corresponding author, and a faculty of Shanghai Institute of Intelligent Electronics & Systems, Shanghai Institute for Advanced Communication and Data Science, and Engineering Research Center of Cyber Security Auditing and Monitoring, Ministry of Education, China.

REFERENCES

[1] 2020. PoC Exploits Do More Good Than Harm: Threatpost Poll. https://threatpost.com/poc-exploits-do-more-good-than-harm-threatpost-poll/152053/.

[2] 2021. How Many Websites Are There in 2021? https://websitesetup.org/news/how-many-websites-are-there/.

[3] 2021. Is It OK to Publish PoC Exploits for Vulnerabilities and Patches? https://www.helpnetsecurity.com/2021/05/05/publishing-poc-exploits/.

[4] 2021. The Invicti AppSec Indicator Spring 2021 Edition: Acunetix Web Vulnerability Report. https://www.acunetix.com/white-papers/acunetix-web-application-vulnerability-report-2021/#another-victim-of-covid-19-web-application-security.

[5] 2022. Acunetix: PHP Security. https://www.acunetix.com/websitesecurity/php-security-1/.

[6] 2022. Common Vulnerabilities and Exposures. https://cve.mitre.org/.

[7] 2022. CVE-2018-15139 in NVD. https://nvd.nist.gov/vuln/detail/CVE-2018-15139.

[8] 2022. How to Secure PHP Web Applications and Prevent Attacks? https://docs.php.earth/security/intro/.

[9] 2022. National Vulnerability Database. https://nvd.nist.gov/.

[10] 2022. Openwall. http://www.openwall.com/.

[11] 2022. OWASP Community. https://owasp.org/www-community/.

[12] 2022. PHP Manual. https://www.php.net/manual/zh/index.php.

[13] 2022. PHP-Parser Source Code. https://github.com/nikic/PHP-Parser.

[14] 2022. ReDebug Source Code. https://github.com/dbrumley/redebug.

[15] 2022. Release of AFV. https://github.com/seclab-fudan/AFV.

[16] 2022. Securityfocus. https://www.securityfocus.com/vulnerabilities.

[17] 2022. Universal Ctags Source Code. https://github.com/universal-ctags/ctags.

[18] 2022. V-SZZ Source Code. https://figshare.com/ndownloader/files/31748777.

[19] 2022. V0Finder Source Code. https://github.com/WOOSEUNGHOON/V0Finder-public.

[20] 2022. Wala: The T. J. Watson Libraries for Analysis. http://wala.sourceforge.net.

[21] 2022. Website Hacking Statistics You Should Know.

[22] 2022. Xdebug. http://xdebug.org/.

[23] Giovanni Agosta, Alessandro Barenghi, Antonio Parata, and Gerardo Pelosi. 2012. Automated Security Analysis of Dynamic Web Applications through Symbolic Code Execution. In *Proceedings of the 9th International Conference on Information Technology New Generations (ITNG)*. 189–194.

[24] Abeer Alhuzali, Birhanu Eshete, Rigel Gjomemo, and VN Venkatakrishnan. 2016. Chainsaw: Chained Automated Workflow-based Exploit Generation. In *Proceedings of the 23rd ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 641–652.

[25] Abeer Alhuzali, Rigel Gjomemo, Birhanu Eshete, and VN Venkatakrishnan. 2018. NAVEX: Precise and Scalable Exploit Generation for Dynamic Web Applications. In *Proceedings of the 27th USENIX Security Symposium (USENIX Security)*. 377–392.

[26] Michael Backes, Konrad Rieck, Malte Skoruppa, Ben Stock, and Fabian Yamaguchi. 2017. Efficient and Flexible Discovery of PHP Application Vulnerabilities. In *Proceedings of the 2nd IEEE European Symposium on Security and Privacy (EuroS&P)*. 334–349.

[27] Daniel Costa, Shane McIntosh, Weiyi Shang, Uirá Kulesza, Roberta Coelho, and Ahmed E. Hassan. 2016. A Framework for Evaluating the Results of the SZZ Approach for Identifying Bug-Introducing Changes. *IEEE Transactions on Software Engineering (TSE)* (10 2016), 1–1.

[28] Johannes Dahse and Thorsten Holz. 2014. Simulation of Built-in PHP Features for Precise Static Code Analysis. In *Proceedings of the 21st ISOC Network and Distributed System Security Symposium (NDSS)*. 23–26.

[29] Johannes Dahse and Thorsten Holz. 2014. Static Detection of Second-Order Vulnerabilities in Web Applications. In *Proceedings of the 23rd USENIX Security Symposium (USENIX Security)*. 989–1003.

[30] Jiarun Dai, Yuan Zhang, Hailong Xu, Haiming Lyu, Zicheng Wu, Xinyu Xing, and Min Yang. 2021. Facilitating Vulnerability Assessment through PoC Migration. In *Proceedings of the 28th ACM SIGSAC Conference on Computer and Communications Security (CCS)*.

[31] 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 *Proceeding of the 28th USENIX Security Symposium (USENIX Security)*. 869–885.

[32] Adam Doupé, Ludovico Cavedon, Christopher Kruegel, and Giovanni Vigna. 2012. Enemy of the State: A State-Aware Black-Box Web Vulnerability Scanner. In *Proceeding of the 21st USENIX Security Symposium (USENIX Security)*. 523–538.

[33] 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.

[34] Jiyong Jang, Abeer Agrawal, and David Brumley. 2012. ReDeBug: Finding Unpatched Uode Clones in Entire OS Distributions. In *Proceedings of the 33rd IEEE Symposium on Security and Privacy (S&P)*. 48–62.

[35] Nenad Jovanovic, Christopher Kruegel, and Engin Kirda. 2006. Pixy: A Static Analysis Tool for Detecting Web Application Vulnerabilities. In *Proceedings of the 27th IEEE Symposium on Security and Privacy (S&P)*. 6 pp.–263.

[36] Soheil Khodayari and Giancarlo Pellegrino. 2021. JAW: Studying Client-side CSRF with Hybrid Property Graphs and Declarative Traversals. In *Proceedings of the 30th USENIX Security Symposium (USENIX Security)*. 2525–2542.

[37] 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)*.

[38] Sunghun Kim, Thomas Zimmermann, Kai Pan, and E. Jr. 2006. Automatic Identification of Bug-Introducing Changes. In *Proceedings of the 21th ACM/IEEE International Conference on Automated Software Engineering (ASE)*.

[39] 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.

[40] 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 ISOC Network and Distributed System Security Symposium (NDSS)*.

[41] Ahmed E. Hassan Lingfeng Bao, Xin Xia and Xiaohu Yang. 2022. V-SZZ: Automatic Identification of Version Ranges Affected by CVE Vulnerabilities. In *Proceedings of the 44th ACM/IEEE International Conference on Software Engineering (ICSE)*.

[42] M.tech.Scholar. 2016. To Enhance Type 4 Clone Detection in Clone Testing. *International Journal of Computer Science and Information Technologies (IJCSIT)* (2016).

[43] 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 *Proceeding of the 27th USENIX Security Symposium (USENIX Security)*. 919–936.

[44] Hamid Nasirloo and Fatemeh Azimzadeh. 2018. Semantic Code Clone Detection using Abstract Memory States and Program Dependency Graphs. In *Proceedings of the 4th International Conference on Web Research (ICWR)*. 19–27.

[45] 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.

[46] Benjamin Nielsen, Behnaz Hassanshahi, and François Gauthier. 2019. Nodest: Feedback-driven Static Analysis of Node.js Applications. In *Proceedings of the 27th Joint Meeting on Foundations of Software Engineering (FSE)*.

[47] Sunnyeo Park, Daejun Kim, Suman Jana, and Sooel Son. 2022. FUGIO: Automatic Exploit Generation for PHP Object Injection Vulnerabilities. In *Proceeding of the 31st USENIX Security Symposium (USENIX Security)*.

[48] Giancarlo Pellegrino, Martin Johns, Simon Koch, Michael Backes, and Christian Rossow. 2017. Deemon: Detecting CSRF with Dynamic Analysis and Property Graphs. In *Proceedings of the 24th ACM SIGSAC Conference on Computer and Communications Security (CCS)*.

[49] Giancarlo Pellegrino, Constantin Tschürtz, Eric Bodden, and Christian Rossow. 2015. jÄk: Using Dynamic Analysis to Crawl and Test Modern Web Applications. In *Proceedings of the 18th International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*. 295–316.

[50] Dawei Qi, Hoang DT Nguyen, and Abhik Roychoudhury. 2013. Path Exploration Based on Symbolic Output. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 22, 4 (2013), 1–41.

[51] R. Bhatia R. Tekchandani and M. Singh. 2018. Semantic Code Clone Detection for Internet of Things Applications using Reaching Definition and Liveness Analysis. *The Journal of Supercomputing* 74, 9 (2018), 4199–4226.

[52] Orpheas van Rooij, Marcos Antonios Charalambous, Demetris Kaizer, Michalis Papaevripides, and Elias Athanasopoulos. 2021. WebFuzz: Grey-Box Fuzzing for Web Applications. In *Proceedings of the 26th European Symposium on Research in Computer Security (ESORICS)*. 152–172.

[53] Hitesh Sajnani, Vaibhav Saini, Jeffrey Svajlenko, Chanchal K Roy, and Cristina V Lopes. 2016. SourcererCC: Scaling Code Clone Detection to Big-Code. In *Proceedings of the 38th International Conference on Software Engineering (ICSE)*. 1157–1168.

[54] Luis Alberto Benthin Sanguino and Rafael Uetz. 2017. Software Vulnerability Analysis using CPE and CVE. *arXiv preprint arXiv:1705.05347* (2017).

[55] Ridwan Shariffdeen, Xiang Gao, Gregory Duck, Shin Hwei Tan, Julia Lawall, and Abhik Roychoudhury. 2021. Automated Patch Backporting in Linux (Experience Paper). In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. 633–645.

[56] Ridwan Salihin Shariffdeen, Shin Hwei Tan, Mingyuan Gao, and Abhik Roychoudhury. 2021. Automated Patch Transplantation. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 30, 1 (2021).

[57] Jacek Śliwerski, Thomas Zimmermann, and Andreas Zeller. 2005. When Do Changes Induce Fixes? *ACM SIGSOFT Software Engineering Notes (SEN)* 30, 4 (2005), 1–5.

[58] Fangqi Sun, Liang Xu, and Zhendong Su. 2011. Static Detection of Access Control Vulnerabilities in Web Applications.. In *Proceeding of the 20th USENIX Security Symposium (USENIX Security)*, Vol. 64.

[59] Minh-Thai Trinh, Duc-Hiep Chu, and Joxan Jaffar. 2014. S3: A Symbolic String Solver for Vulnerability Detection in Web Applications. In *Proceedings of the 21st ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 1232–1243.

[60] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. 1999. Soot: A Java Bytecode Optimization Framework. In *Proceedings of the 9th Conference of the Centre for Advanced Studies on Collaborative Research (CASCON)*. 214–224.

[61] Pengcheng Wang, Jeffrey Svajlenko, Yanzhao Wu, Yun Xu, and Chanchal K Roy. 2018. CCAligner: A Token Based Large-Gap Clone Detector. In *Proceedings of the 40th International Conference on Software Engineering (ICSE)*. 1066–1077.

[62] Xinda Wang, Shu Wang, Pengbin Feng, Kun Sun, and Sushil Jajodia. 2021. PatchDB: A Large-Scale Security Patch Dataset. In *Proceedings of the 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 149–160.

[63] Seongil Wi, Sijae Woo, Joyce Jiyoung Whang, and Sooel Son. 2022. HiddenCPG: Large-Scale Vulnerable Clone Detection Using Subgraph Isomorphism of Code Property Graphs. In *Proceedings of the 31st ACM Web Conference(WWW)*. 755–766.

[64] Seunghoon Woo, Dongwook Lee, Sunghan Park, Heejo Lee, and Sven Dietrich. 2021. V0Finder: Discovering the Correct Origin of Publicly Reported Software Vulnerabilities. In *Proceedings of the 30th USENIX Security Symposium (USENIX Security)*. 3041–3058.

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

[66] Fang Yu, Muath Alkhalaf, Tevfik Bultan, and Oscar H Ibarra. 2014. Automata-Based Symbolic String Analysis for Vulnerability Detection. *Formal Methods in System Design* 44, 1 (2014), 44–70.

[67] Su Zhang, Xinming Ou, and Doina Caragea. 2020. Automated CPE Labeling of CVE Summaries with Machine Learning. In *Proceeding of 20th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*.

[68] Yunhui Zheng and Xiangyu Zhang. 2013. Path Sensitive Static Analysis of Web Applications for Remote Code Execution Vulnerability Detection. In *Proceedings of the 35th International Conference on Software Engineering (ICSE)*. 652–661.

[69] Yunhui Zheng, Xiangyu Zhang, and Vijay Ganesh. 2013. Z3-str: A Z3-Based String Solver for Web Application Analysis. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering (FSE)*. 114–124.