URadar: Discovering Unrestricted File Upload Vulnerabilities via Adaptive Dynamic Testing

Yuanchao Chen, Yuwei Li, Zulie Pan, Yuliang Lu, Juxing Chen, and Shouling Ji, *Member, IEEE*

***Abstract*— Unrestricted file upload (UFU) vulnerabilities, especially unrestricted executable file upload (UEFU) vulnerabilities, pose severe security risks to web servers. For instance, attackers can leverage such vulnerabilities to execute arbitrary code to gain the control of a whole web server. Therefore, it is significant to develop effective and efficient methods to detect UFU and UEFU vulnerabilities. Towards this, most state-of-the-art methods are designed based on dynamic testing. Nevertheless, they still entail two critical limitations. 1) They heavily rely on manual efforts, which are error-prone and have poor adaptability. 2) They seldom leverage effective information to guide the testing, resulting in generating a large number of invalid test cases. Such limitations severely hinder the performance of UFU vulnerability detection. In this paper, we propose URadar, an adaptive dynamic testing-based method for detecting UFU and UEFU vulnerabilities. There are three core designs in URadar, including *file upload interface identification*, *file type restriction inference*, and *invalid mutation combination filtration*, which can effectively solve the two limitations of existing methods. To evaluate the performance of URadar, we conduct extensive experiments and compare URadar with state-of-the-art methods (e.g., FUSE, RIPS). In testing 18 web applications, URadar discovers 26 UEFU vulnerabilities, where 8 are new, and 6 have been assigned new CVE/CNNVD IDs. By contrast, FUSE and RIPS find 14 and 2 UEFU vulnerabilities, respectively. To discover the same number of UFU vulnerabilities, FUSE needs to send 73,261 request packets with a time cost of 2,791.1s on average, 23.43 and 20.53 times of the requirements for URadar. The above results demonstrate that URadar significantly outperforms the state-of-the-art methods. In addition, we have open-sourced URadar to facilitate future research on UFU vulnerability detection.**

***Index Terms*— Web security, file upload vulnerability, dynamic testing, cross-location, interface identification.**

I. Introduction

With the rapid development of the Internet, web applications have been integrated into numerous aspects of people's lives. File upload is a prevalent functionality supported by most web applications, such as social media and forums. It allows users to upload various types of files, including figures, documents, audio, and video, making it convenient for users to save or share data resources.

However, the file upload functionality may pose severe security risks to the web server. Due to the flaws in web security checks, an attacker may upload a malicious file that can bypass those checks, which are referred to as unrestricted file upload (UFU) vulnerabilities [1]. Even worse, if the attacker can operate the executable code of such an uploaded file via a URL, these UFU vulnerabilities evolve into unrestricted executable file upload (UEFU) vulnerabilities[1] [2]. Attackers can leverage UEFU vulnerabilities to execute arbitrary code to gain the control of a whole web server [3]. As UFU vulnerabilities can cause severe security threats, OWASP [4] considers UFU vulnerabilities as the top dangerous web vulnerabilities. Consequently, how to detect UFU vulnerabilities effectively and efficiently is of paramount importance to web application security.

Existing UFU vulnerability detection methods [2], [5], [6], [7], [8] are mainly based on static or dynamic testing. Static program analysis method mainly analyzes the source code of the file upload operation to check whether the parameters in the file upload operation function come from untrustworthy sources [5], [6]. However, due to the dynamics, diversity, complexity, and flexibility of web scripting languages (e.g., PHP), static program analysis has limitations in characterizing the UFU vulnerabilities. For instance, static taint data flow can be easily broken, resulting in false negatives, and it is challenging to use for analyzing the user-defined sanitization actions, leading to false positives [9].

Compared to static program analysis, dynamic testing has lower false positives as it can really trigger the vulnerabilities in practice. It detects such vulnerabilities based on the file upload results by generating a large number of test cases and simulating the actual upload process. For instance, FUSE [2] is a representative dynamic testing method with 13 carefully designed mutation operations. UFuzzer [8] combines static program analysis and dynamic testing, in which it utilizes static program analysis to create testing templates for dynamic testing. However, the state-of-the-art methods for UFU and UEFU vulnerability detection still face the following two main challenges.

Challenge 1: How to reduce the human efforts to make the testing more automatic and efficient? The existing methods

Manuscript received 12 June 2023; revised 15 October 2023 and 6 November 2023; accepted 12 November 2023. Date of publication 21 November 2023; date of current version 1 December 2023. This work was supported in part by the National Key R&D Program of China under Grant 2021YFB3100500 and in part by NSFC under Grant 62202484. The associate editor coordinating the review of this manuscript and approving it for publication was Dr. Z. Berkay Celik. *(Corresponding authors: Yuwei Li; Zulie Pan; Yuliang Lu.)*

Yuanchao Chen, Yuwei Li, Zulie Pan, Yuliang Lu, and Juxing Chen are with the College of Electronic Engineering, National University of Defense Technology, Hefei 230037, China (e-mail: liyuwei@nudt.edu.cn; panzulie17@nudt.edu.cn; luyuliang@nudt.edu.cn).

Shouling Ji is with the College of Computer Science and Technology, Zhejiang University, Hangzhou 310027, China.

Digital Object Identifier 10.1109/TIFS.2023.3335885

[1]UFU vulnerabilities contain UEFU vulnerabilities.

still heavily rely on human efforts to identify the upload interface and construct the input templates. These tasks are tedious and error-prone, which make them hard to apply to more web applications. For instance, FUSE requires manually constructing the configuration files, which needs manually finding the upload interface. This method is inefficient and may lead to false negatives. On the other hand, modern web applications have complex and various structures and formats, which bring many challenges to make the above process automatic.

Challenge 2: How to reduce the blindness of the testing? The existing methods seldom leverage effective information (e.g., sanitization actions information) in the code to guide the testing, resulting in the generation of a large number of invalid test cases, which is inefficient. For instance, web applications usually define some checks on the inputs. Without effective code information as guidance, most of the randomly generated inputs can only trigger the superficial code of the web application under test, which are invalid. Thus, it is necessary to study how to extract valuable code information and use them to guide the testing.

To address the above challenges, in this paper, we propose URadar, an adaptive dynamic testing method for discovering UFU vulnerabilities. There are three core designs in URadar, namely, *file upload interface identification*, *file type restriction inference*, and *invalid mutation combination filtration*. To overcome *challenge 1*, we propose the design of *file upload interface identification*. Specifically, employed with a state-aware page traversal method, URadar is able to automatically identify as many as possible upload interfaces and constructs HTTP request templates for file upload. To overcome *challenge 2*, we propose the designs of *file type restriction inference* and *invalid mutation combination filtration*. URadar uses the extension list cross-location to find the sanitization action information for the file upload function. Then, URadar uses this sanitization action information to guide the generation of test cases, with the goal of improving the test case effectiveness during testing. Using the invalid mutation combination filtration method, URadar filters invalid mutation combinations of non-executable file properties, significantly reducing the generation of invalid test cases. Notably, URadar utilizes the parsing rules for each PHP version to determine whether the uploaded file can be parsed and executed. With the design, URadar can identify UFU vulnerabilities in web applications that support multiple PHP versions without building various PHP version test environments.

We conduct comprehensive experiments to verify the effectiveness and efficiency of URadar and compare it with existing state-of-the-art tools (e.g., FUSE, RIPS) for UFU vulnerability detection. Regarding effectiveness, the experimental results show that URadar can find 26 UEFU vulnerabilities in 18 popular web applications, including eight new UEFU vulnerabilities, six of which are assigned new CVE/CNNVD IDs. In contrast, FUSE finds 14 UEFU vulnerabilities in these web applications, and RIPS finds 2. To assess efficiency, we compare URadar and FUSE in the same state. To discover the same number of UFU vulnerabilities, FUSE needs to send 73,261 request packets and consumes 2,791.1s on average, which are 23.43 and 20.53 times of the corresponding requirements for URadar.

The main contributions of this paper are summarized as follows.

- **Novel Methods.** We propose URadar, a novel UFU vulnerability detection framework based on adaptive dynamic testing. There are three core designs in URadar: *file upload interface identification*, *file type restriction inference*, and *invalid mutation combination filtration*.
- **SOTA Level Performance.** We conduct comprehensive experiments to evaluate the performance of URadar. Compared to state-of-the-art methods (e.g., FUSE and RIPS), URadar has significant outstanding performance.
- **Open-sourced Framework.** To support the advancement of future research in the detection of UFU and UEFU vulnerabilities, we open-sourced URadar.[2]

II. Background: UFU Vulnerabilities

Fig. 1 illustrates the file upload process of a PHP web application. First, a user sends a file upload request packet that contains the file to be uploaded via the interface provided by the web application (❶). The uploaded file is initially saved in a temporary directory. Then, the web application usually performs security checks on the uploaded file (❷). In the example in Fig. 1, the web application checks whether the uploaded file extension appears on the blacklist to determine whether the uploaded file is allowed. If the file passes the security checks, it will be saved on the server (❸). Subsequently, the user can access the uploaded file through the return URL (❹).

However, improper security checks may lead to UFU vulnerabilities. An attacker can exploit such vulnerabilities to upload malicious files, such as webshells to compromise the web server. For instance, as shown in ❹, after a malicious file that contains executable code is uploaded successfully by an attacker, he/she can send an instruction to the uploaded file, such as `'?pass=phpinfo();'`. If this code can be executed successfully on the web server, the attacker can obtain web server environment information. Additionally, the potential implications of UFU vulnerabilities are varied and serious. They can result in actions such as uploading a large file which will cause a spatial denial of service, leaking sensitive data, overwriting files on the server, or even facilitating privilege escalation. All of these scenarios pose severe threats to the security of web applications. Thus, designing an efficient method to detect UFU vulnerabilities is paramount.

III. System Design

In this section, we first present an overview of URadar. Then, we detail its core designs.

A. Overview

Fig. 2 presents an overview of URadar, composed of five main modules: file upload interface identification, file type

[2]https://github.com/Cyc1e183/URadar

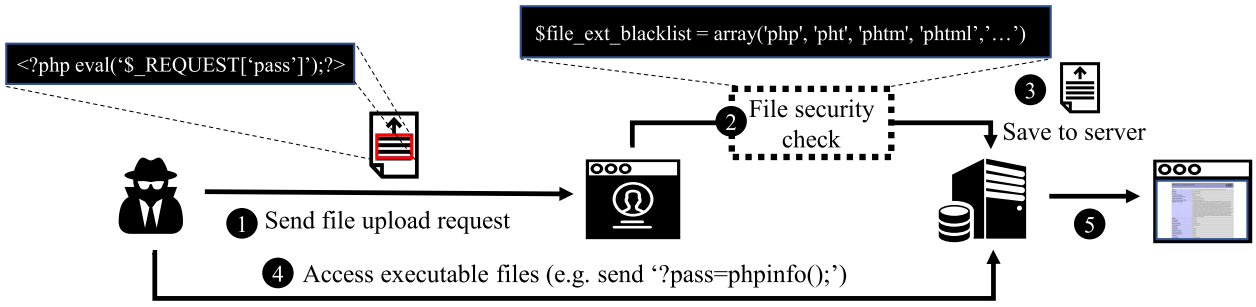


Fig. 1. The file upload process of a PHP web application.

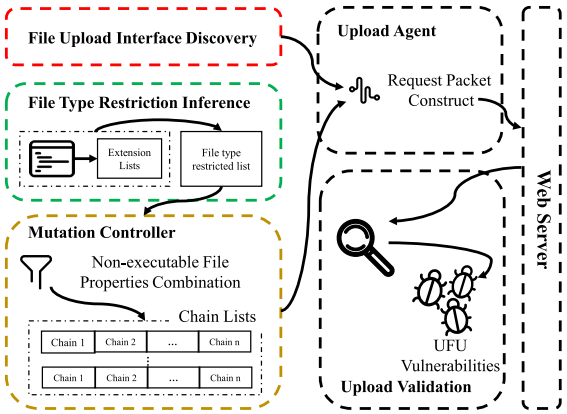


Fig. 2. Overview of URadar.

restriction inference, mutation controller, upload agent, and upload validation.

1) File Upload Interface Identification: This module aims to automatically identify as many file upload interfaces of target web applications as possible. It facilitates request packet construction and comprehensive testing and improves the detection performance for UFU and UEFU vulnerabilities in web applications. To this end, we propose a state-aware page traversal method.

2) File Type Restriction Inference: This module finds the list of file type restrictions of the file upload form. This list in the source code either forbids files from being uploaded or allows only files on the list to be uploaded. Based on the file type restriction list, which is found or generated via the cross-location method, URadar uses it as a guide to enhance the validity of test cases.

3) Mutation Controller: The functionality of this module is to construct a mutation chain based on the file restriction inference results and the combination rules for non-executable file properties. Here, the mutation chain represents a list of mutation operations. During the dynamic testing process, new test cases are generated by mutating the seed files through the operations in the mutation chain.

4) Upload Agent: The upload agent module completes the following tasks. First, based on the upload interface discovery results, URadar constructs an initial file upload request. Then, it mutates the key components of the request packet, according to the operations in the mutation chain. Finally, it adds a unique identifier to the upload request packet for confirmation in the next step and sends it to the target web application.

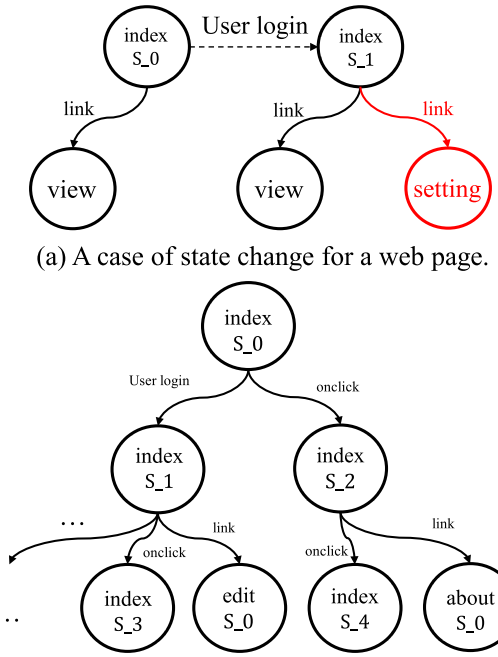
5) Upload Validation: After the file upload HTTP request packet is sent, this module verifies whether the request can trigger UFU and UEFU vulnerabilities. Specifically, it first uses the unique identifier injected by the upload agent or the hash of the file to determine whether the file is successfully uploaded to the target web server. Then, it tests the uploaded file through the return URL to determine whether the code in the file can be executed. We insert code into the PHP seed file that can generate a fixed identification string after execution, e.g., the code "$sign= hex2bin('5552616461725f436f64655f457865637-574696f6e');". When we access the file via a URL, if the code in the file can be parsed and executed, we will obtain a fixed string "URadar_Code_Execution".

Notably, it carries out cross-PHP-version detection for both UFU and UEFU vulnerabilities. Web applications usually support multiple PHP versions. We have analyzed the parsing rules of each version of PHP and found some differences among different versions. For example, the parsing rule for PHP 5.6 and PHP 7.0 is "^.$ph(p[3457]?|t|tml|ps)$$" [10], [11], which can parse files with extensions php, php3, php4, php5, php7, phpt, phtml, and phps. The parsing rule for PHP 7.2 and above is "^.$ph(ar|php|phps|phtml)$$" [12], which removes the parsing of "php[3457]" and "pht" extension files but adds the parsing of "phar" extension files. To avoid false negatives due to differences in PHP versions, this module is compatible with the parsing rules for each version of PHP. When a file conforming to the PHP parsing rules can be successfully uploaded, whereas the fixed string "URadar_Code_Execution" is not obtained when accessing the file. In this condition, it is necessary to further assess whether the web application has compressed the file's content, that is, whether the hash of the file is unchanged. Next, URadar check the ".htaccess" file in the current folder, which does not forbid PHP code parsing. That is, URadar can identify UFU vulnerabilities for web applications that support multiple PHP versions while avoiding false negatives caused by them.

In the following, we present a detailed description of the core designs of URadar, including *file upload interface identification*, *file type restriction inference*, and *invalid mutation combination filtration*.

B. File Upload Interface Identification

Different from the input interface of a binary program, the data interaction interface of a web application usually exists



(a) A case of state change for a web page.

(b) A case of web page state traversal.

Fig. 3. A case of state change and state traversal of web pages.

mainly on the web application page. Users commonly engage with server-side web applications by interacting with various elements on the page, such as clicking controls, completing forms, and more. Given that web applications cannot guarantee sufficient security detection capabilities for user-inputted data on every interactive interface, increasing the coverage of these interfaces can enhance the likelihood of detecting vulnerabilities. Therefore, the broader the coverage of interactive interfaces in a web application, the higher the chance of triggering and identifying vulnerabilities. The coverage rate of file upload interfaces is a key issue in web application UFU and UEFU vulnerability detection. To enhance its coverage rate, URadar employs a state-aware page traversal strategy, identifying the file upload interface and constructing the HTTP request packet template grounded on the interface's information. This approach is dissected into three principal components: the traversal process, interface identification, and the construction of the request packet template. Each part will be introduced thoroughly.

1) Traversal: Traversing the target web applications as comprehensively as possible is the primary challenge in identifying the file upload interface. Due to the dynamic and interactive nature of the existing web front-end, web pages possess widely diverse features. When the state changes, different content may appear on the same web page (i.e., some components may appear only when a specific state is triggered). For example, Fig. 3 (a) presents a case of state change for a web page, where each circle represents a unique state, and each link represents an action. The action `user login` transforms the state of the web page `index` from S_0 to S_1. Importantly, the link to `setting` only appears in state S_1. Therefore, it is crucial to capture the state changes of every web page during traversal. Otherwise, some web pages will not be visited, and even worse, some interfaces may be missed, hindering the comprehensive detection of UFU vulnerabilities.

URadar traverses the target web application by a breadth-first traversal method starting from the root web page. When URadar accesses a web application, it first builds a DOM tree model based on the HTML elements of the current web page. This DOM tree model consists of triples in the form of $< URL\_link, Form, Js\_action >$. Here `URL_link` represents the link for the web application on the page in the current state. `Form` refers to the form information on the page. `Js_action` represents a JavaScript event present on the page. After any operation is performed on the web page, URadar rebuilds the DOM tree model of the page and judges whether the state of the page has changed based on the difference in the triples. Using Fig. 3 (b) as an example, the S_0 state of the `index` page is the root state, from which URadar discovers two new page states, S_1 and S_2, of the `index` page after submitting a form and triggering an "onclick" event. By traversing the S_1 and S_2 states of the `index` page, URadar can discover some new page states from the S_1 state of the `index` page, such as the S_3 state of the `index` page and the S_0 state of the `edit` page. Likewise, URadar can discover the S_4 state of the `index` page and the S_0 state of the `about` page from the S_2 state of the `index` page. URadar continues to traverse the web application by exploring newly discovered page states until no additional page states are found, trying to complete a comprehensive traversal of the entire application.

2) Interface Identification: URadar identifies the file upload interfaces at each state node while traversing a web application. As we have found by investigating the 18 web applications listed in Table I, the file upload interfaces usually reside in the source code of the web pages. Listing 1 shows the HTML code of a file upload form in WordPress [13]. As illustrated by this sample code, a file upload form consists of the following properties: (1) `<form>...</form>`, (2) `<input type="file"`, and (3) `<input type="submit"`. When URadar encounters a state node, the identification module first analyzes the source code of the web page, looking for the presence of the `<form></form>` element. It then uses this information to ascertain whether a form exists where user data can be inputted. Next, it searches for the existence of a `type="file"` element to determine whether this form is a file upload form. Finally, URadar checks whether a `type="submit"` element is present at the end of the form to determine the form's availability on the page.

3) Request Packet Template Construction: Based on the identified file upload form on the web page, URadar can construct the file upload HTTP request packet template to generate numerous test cases. The method is leveraging the HTML information of the file upload form, such as the Listing 1, to construct upload request packets. In this form, the `method` means that the request method (mainly includes GET and POST), the `action` means that the request address path and the `<input>` tags contain the parameter settings and the corresponding values in the file upload request packet. In this approach, the contents of the 'id' or 'name'

```
<form method="post" enctype="multipart/form-data" class=
"wpupload-form" action="/wp-admin/update.php?action=upl-
oad-plugin">
 <input type="hidden" id="_wpnonce" name="_wpnonce" val-
 ue="78dcb1b448"/>
 <input type="hidden" name="_wp_http_referer" value="/w-
 padmin/plugin-install.php" />
 <input type="file" id="pluginzip" name="pluginzip"/>
 <input type="submit" name="install-plugin-submit" id="
 install-plugin-submit"
 class="button" value="Install Now"/>
</form>
```

Listing. 1: File upload form code in WordPress.

```
------WebKitFormBoundaryclXHhOTuZe2F0XPB
Content-Disposition: form-data; name="_wpnonce"

78dcb1b448
------WebKitFormBoundaryclXHhOTuZe2F0XPB
Content-Disposition: form-data; name="_wp_http_referer"

/wp-admin/plugin-install.php
------WebKitFormBoundaryclXHhOTuZe2F0XPB
Content-Disposition: form-data; name="pluginzip"; filename=
"@file"
Content-Type: @file_type

@file_content
------WebKitFormBoundaryclXHhOTuZe2F0XPB
Content-Disposition: form-data; name="install-plugin-submit"

Install Now
------WebKitFormBoundaryclXHhOTuZe2F0XPB--
```

Listing. 2: Example of a file upload request.

fields in `type="hidden"` and `type="submit"` elements are merged with the corresponding `value` contents according to the content of `enctype`. Additionally, the contents of `type="file"` are replaced with tag characters (for example, the string "@file") to facilitate subsequent mutations. Thus far, we can get the information for constructing the file upload request packet. Take the upload form information in Listing 1 as an example. We can get the URL of the file upload HTTP request packet as "http://ip:port/wp-admin/update.php?action=upload-plugin", and the HTTP body is shown in Listing. 2. This method can generate request packets corresponding to the file upload forms for the 18 web applications in Table.

C. File Type Restriction Inference

For the mutation operations on uploaded files, to ensure standard semantics of the files, the current mutation methods are based on the file extension, the `content_type` values in the upload request packets, and the file content. It means that the mutation process will pretend to upload certain types of files admitted by the website under test. A file upload form on a website typically uses either a user-defined blacklist or a whitelist to limit the types of files that can be uploaded. That is, blacklists and whitelists are commonly used sanitization actions for the file upload functionality to ensure security. For example, for avatar upload, only the uploading of image-type files may be enabled, such as JPG, PNG, and GIF. Only compressed files such as ZIP, RAR, and TAR.GZ files may be allowed for plugin upload. In contrast, for the uploading of personal resumes, it may be possible to upload only text-type files, such as PDF and DOC, while executable-type files are disallowed. The code for the plugin upload in Joomla [14] is shown in Listing 3.

```
1 #administrator\components\com_installer\models\install.php
2 <?php
3 ...
4 protected function _getPackageFromUpload(){
5     ...
6     $userfile = $input->files->get('install_package', null,
7     'raw');
8     ...
9 }
10 ?>
11 # libraries\fof\input\jinput\files.php
12 <?php
13 ...
14 public function get($name, $default = null, $filter = 'cmd')
15 {
16     ...
17     $isSafe = JFilterInput::isSafeFile($results);
18     if (!$isSafe){
19         ...
20     }
21 }
22 ?>
23 #libraries\src\Filter\InputFilter.php
24 <?php
25 ...
26 public static function isSafeFile($file, $options = array())
27 {
28     ...
29     'php_ext_content_extensions' => array('zip', 'rar',
30     'tar', 'gz', 'tgz', 'bz2', 'tbz', 'jpa'),
31     ...
32 }
33 ?>
```

Listing. 3: Code for plug-in upload in Joomla.

As lines 6 and 17 in Listing 3 show, when a user uploads a plug-in in the background console of Joomla, it will detect whether the extension of the uploaded file is present in the list "php_ext_content_extensions". For example, line 29 of the code shows that Joomla allows only files with extensions of compressed package types such as ZIP to be uploaded. Therefore, during the dynamic testing process, only the mutations on such file extensions are valid, while others are invalid. Similarly, if a web application restricts MIME (Multipurpose Internet Mail Extensions) [15] types, all mutations except the allowed MIME mutations types are considered invalid. The MIME type indicates the nature and format of a document, file, or assortment of bytes. If URadar cannot find these restrictions in time, it will traverse the mutation search space completely before terminating the testing. As a result, a large number of invalid test packets will be sent, seriously affecting testing efficiency.

Therefore, we propose an efficient file type restriction inference method based on file type list cross-location. The workflow for file type restriction inference is as follows. First, URadar abstracts all possible file extension restriction lists from the web application source code. Then URadar accurately determines the restriction list for each file upload interface by identifying the inclusion and exclusion sets.

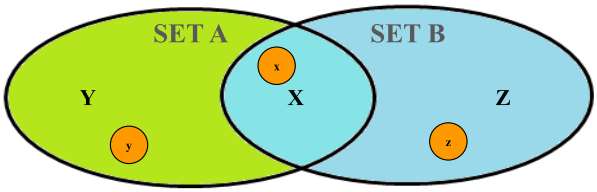


Fig. 4. Set relationships for $\mathbf{A} \cap \mathbf{B} \neq \emptyset$.

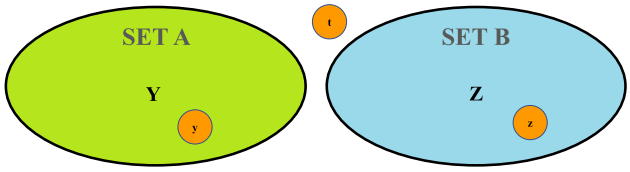


Fig. 5. Set relationships for $\mathbf{A} \cap \mathbf{B} = \emptyset$.

1) Information Extraction: As shown in Listing 3, lists that restrict the extensions of uploaded files usually exist in the source code of a web application. Moreover, such a list may also appear in a web application's configuration and database files, such as the ".conf" files and ".sql" files. URadar strives to cover all file extension restriction lists in the web application's source code as comprehensively as possible. This ensures that the list for the current file upload form can be accurately located and generated using the cross-location method. For information extraction, URadar maintains an extensive and comprehensive file extension database, which is used to extract and construct the file type restriction list. First, URadar traverses each file in the target web application and uses string analysis to determine whether the current file has a file extension string. Second, URadar analyzes the strings before and after an identified file extension string and the context of the file extension string. If the string before and after the file extension string is also an extension string, this line is identified as a file type list, such as the code on line 29 in Listing 3. If the strings before and after the file extension string are not extension strings but file extension string appears multiple times in the context, these file extensions are extracted to form a file extension list. Finally, URadar cleans the extracted file extension list data to remove non-file extension strings extracted by mistake to prevent this invalid string information from affecting subsequent testing.

2) File Type List Cross-Location: For the extracted file extension lists, we must determine which list restricts a file upload form and whether the restriction list is a blacklist or a whitelist. We treat each extracted list as a set to facilitate inter-set operations for identifying inclusion/exclusion sets. For example, consider two file extension sets $\mathbf{A}$ and $\mathbf{B}$, where $\mathbf{A} \cap \mathbf{B} \neq \emptyset$. We perform intersection and difference operations on sets $\mathbf{A}$ and $\mathbf{B}$, namely, $\mathbf{X} = \mathbf{A} \cap \mathbf{B}$, $\mathbf{Y} = (\mathbf{A} - \mathbf{B})$ and $\mathbf{Z} = (\mathbf{B} - \mathbf{A})$, as shown in Fig. 4. First, we take a file with extension x to be uploaded, where $x \in \mathbf{X}$. If the file can be uploaded successfully, it is determined that set $\mathbf{A}$ and set $\mathbf{B}$ are the whitelist. Then, it is necessary to determine which set is the restriction list for the current upload form. To this end, we take files with extensions y and z to be uploaded, where $y \in \mathbf{Y}$ and $z \in \mathbf{Z}$. If the file with extension y can be uploaded successfully but the file with z cannot, this means that the set $\mathbf{A}$ is the file extension restriction list for the current upload form. Conversely, set $\mathbf{B}$ is the restriction list of the current upload form. If both files can be uploaded successfully, then the extension restriction list for the current upload form is the union of sets $\mathbf{A}$ and $\mathbf{B}$, that is, $list = \mathbf{A} \cup \mathbf{B}$. Otherwise, neither set $\mathbf{A}$ nor set $\mathbf{B}$ is the file extension restriction list for the current file upload form.

Furthermore, there may be a situation in which $\mathbf{A} \cap \mathbf{B} = \emptyset$, as shown in Fig. 5. For example, suppose that set $\mathbf{A}$ is judged to be a whitelist and set $\mathbf{B}$ is judged to be a blacklist. URadar then requires further precise judgment. URadar takes a file with extension t, where $t \notin \mathbf{A}$ and $t \notin \mathbf{B}$. When a whitelist restricts the form, files outside the whitelist cannot be uploaded. That is, if the selected file with extension t cannot be successfully uploaded, this indicates that set $\mathbf{A}$ is the exact list of restrictions. Conversely, when a blacklist restricts the form, all files outside the blacklist can be uploaded. Therefore, if the selected file with extension t can be successfully uploaded, this indicates that set $\mathbf{B}$ is the exact list of restrictions.

In accordance with the file extension lists found in or generated from the web application, URadar further determines whether the current file upload point restricts the MIME types corresponding to the file extensions in the list. For blacklist restriction, URadar generates a file with an extension that does not appear in the restriction list and changes the MIME type of the file to a MIME type corresponding to an extension that does appear in the list. Depending on the upload situation for this file, it is determined whether the upload form has MIME-type restrictions. URadar uses this method to determine whether the file restriction list restricts only the file extension, the file MIME type, or both. This method of file extension list cross-location can be extended to the case of multiple sets. Finally, URadar feeds the file type restriction list back to the mutation controller module to restrict the mutation operations. The file type restrictions list may differ for different file upload forms. This method can find or generate a correct file type restriction list for each file upload form.

In the example shown in Listing 3, the file type restriction list is {'zip', 'rar', 'tar', 'gz', 'tgz', 'bz2', 'tbz', 'jpa'}, and it is a whitelist. For example, 'JPG' is not included in the restriction list. Because the file MIME type is not specified in the file type restriction list, files with extensions not on the list are invalid mutations. If the list restricts the file MIME type, either a file extension of ".jpg" or a value of "image/jpeg" for `content_type` in the request packets would be considered an invalid mutation. URadar uses this restriction list to prune all mutation branches related to the JPG file type in the mutation search space. This file type restriction inference method yields the restriction information relevant to the file upload form during the testing process, thereby allowing the search space for the mutation operations to be dynamically adjusted.

D. Invalid Mutation Combination Filtration

As seen on line 7 of Listing 4, WordPress only confirms if an uploaded file's type or extension fulfills the requirements during the file processing stage. It does not concurrently

```
1 #wp-admin\includes\file.php
2 <?php
3 ...
4 function _wp_handle_upload(!&$file,$overrides,$time,$action)
5 {
6     ...
7     if((!$type||!$ext)&&!current_user_can('unfiltered_upload'))
8     {
9       return call_user_func_array($upload_error_handler,
10      array(&$file, __('Sorry, this file type is not permitted
11      for security reasons.')));
12    }
13    ...
14 }
15 ?>
16
17 #wp-includes\functions.php
18 <?php
19 ...
20 function wp_check_filetype_and_ext($file,$filename,$mimes=
21    null){
22    ...
23    $mime_to_ext = apply_filters('getimagesize_mimes_to_exts',
24    array(
25        'image/jpeg' => 'jpg',
26        'image/png'  => 'png',
27        'image/gif'  => 'gif',
28        'image/bmp'  => 'bmp',
29        'image/tiff' => 'tif',));
30    ...
31 }
32 ?>
```

Listing. 4: Code for uploaded file detection in WordPress.

verify its MIME type and extension. This can be seen from lines 7 and 20, as the condition on line 7 is satisfied irrespective of the value of content_type when the file extension is “jpg”, “png”, “gif”, “bmp”, or “tif”. For example, if a web application conducts a security check based solely on the uploaded file’s extension, a file with a “.png” extension but with a `content_type` value of “image/jpeg”, and ZIP file header data added to its content, could be successfully uploaded. Indeed, these test cases generated by combining attributes of non-executable files do not pose a direct security threat to web applications. Hence, they are meaningless. Nonetheless, existing methods generate a large number of such invalid test cases.

Therefore, we propose a method of invalid mutation combination filtration. URadar combines only related mutation operations for the same type of non-executable file in each operation combination involving non-executable file mutation. The method is shown in Algorithm 1. URadar records the file types FileType of the original mutation chain and MuType of the new mutation operation, as well as the file types DetType of the resource file. In the mutation operation chain construction process, it is checked whether the combination of mutation chains involves non-executable files, that is, whether FileType and MuType match file types in DetType. Then, if FileType equals MuType, a new mutation chain will be generated and returned as a result, as shown in Lines 13 and 14. Otherwise, the corresponding mutation combination is judged invalid.

For example, suppose that the mutation of a PHP seed file includes the following mutation operations.

$$M1 : \{M01_{JPG}, M01_{PDF}, M01_{ZIP}\}$$
$$M2 : \{M02_{JPG}, M02_{PDF}, M02_{ZIP}\}$$
$$M3 : \{M03_{JPG}, M03_{PDF}, M03_{ZIP}\}$$

Algorithm 1 Invalid Mutation Combination Filtration Method

1: **function** PROPERTIESCOMBINATION($base\_chain, mutationop, seed\_type, black\_list, Basic\_detection\_file$)
2: **if** $seedtype \not\subseteq mutation.dependency$ **then**
3: **return** $false$
4: **end if**
5: $FileType \leftarrow base\_chain.type$
6: $MuType \leftarrow mutationop.type$
7: $DetType \leftarrow Basic\_detection\_file.type$
8: **if** $MuType \in black\_list$ **then**
9: **return** false
10: **end if**
11: **if** $FileType \in DetType \&\& MuType \in DetType$ **then**
12: **if** $MuType == FileType$ **then**
13: $base\_chain \leftarrow base\_chain + mutationop$
14: **return** $base\_chain$
15: **else**
16: **return** false
17: **end if**
18: **end if**
19: **end function**

M1 is applied to add a file header prefix to the uploaded file content. For example, $M01_{JPG}$ adds the file header prefix of a JPG file before the uploaded file content. M2 injects malicious code into the metadata of three types of files (i.e., JPG, PDF, and ZIP). M3 changes the file type of `content_type` in the upload request packet. We define M1 and M2 as conflicting because M2 will overwrite the content mutated by M1. Therefore, M1 and M2 will not be combined. These three types of mutation operations are based on non-executable file types. These mutations all modify the relevant content in the upload request packet of the seed file to the relevant properties of the non-executable file. If these operations were to be combined without constraints, 28 mutation chains would be generated.

According to the above analysis and Algorithm 1, however, mutations combining the properties of different non-executable file types, such as $M02_{ZIP}\&M03_{JPG}$, are invalid. Generally, among mutation combinations for non-executable file properties, the mutation combination $M02_X\&M03_Y$ of M2 and M3 is deemed valid if and only if X equals Y. All other such mutation combinations are considered invalid. Thus, the combination of `content_type` “image/jpeg” and a ZIP file header’s prefix is considered an invalid mutation. After invalid mutation combination filtration, the final mutation chains formed by M1, M2, and M3 are as follows.

$$CHAIN : \{\emptyset, M01_{JPG}, M01_{PDF}, M01_{ZIP}, M02_{JPG}, M02_{PDF}, M02_{ZIP}, M03_{JPG}, M03_{PDF}, M03_{ZIP}, M01_{JPG}\&M03_{JPG}, M01_{PDF}\&M03_{PDF}, M01_{ZIP}\&M03_{ZIP}, M02_{JPG}\&M03_{JPG}, M02_{PDF}\&M03_{PDF}, M02_{ZIP}\&M03_{ZIP}\}$$

The number of generated test cases is reduced from 28 to 16. Thus, this approach eliminates approximately 42.9% of the input space in this example, which reduces the generation

TABLE I
THE WEB APPLICATIONS FOR TESTING

#	Application	Version	LoC	#	Application	Version	LoC
0	Elgg	2.3.10	96,582	9	XE	1.11.2	145,169
1	Subrion	4.2.1	319,112	10	WordPress	5.0.3	1,249,842
2	Monstra	3.0.4	107,128	11	phpBB	3.2.5	193,720
3	CMSMadeSimple	2.2.9.1	310,316	12	Drupal	8.6.9	306,343
4	ClipperCMS	1.3.3	60,525	13	CMSimple	4.7.7	52,344
5	Bludit	3.8.1	44,372	14	Concrete	8.4.4	32,632
6	MyBB	1.8.19	237,667	15	GetSimpleCMS	3.3.15	45,776
7	Backdrop	1.12.1	73,739	16	SilverStripe	4.3.0	187,628
8	Joomla	3.9.3	33,949	17	Symphony	2.7.7	129,161

of invalid test cases to an extent, thereby improving the test efficiency of the system.

It is possible to avoid blind traversal of the test case search space by combining file type restriction inference and invalid mutation combination filtration. These methods are used to dynamically adjust the strategy of URadar to reduce the generation of invalid test cases. Thus, the efficiency of URadar in discovering UFU vulnerabilities is improved.

IV. EVALUATION

In this section, we evaluate the performance of URadar in discovering UFU vulnerabilities and compare it to state-of-the-art tools. We aim to evaluate URadar by answering the following research questions.

- RQ1. How effective is URadar's augmentation technology in discovering UFU and UEFU vulnerabilities in web applications? How does the number of vulnerabilities found by URadar compare to that of the state-of-the-art tools (e.g., FUSE, RIPS)?
- RQ2. How does URadar's efficiency compare to the dynamic testing state-of-the-art tool (i,e, FUSE) in discovering UFU and UEFU vulnerabilities?
- RQ3. Do the file type restriction inference method and invalid mutation combination filtration method in URadar contribute to fuzzing, and how do they perform?

A. Experimental Setup

1) Benchmark Applications: Table I shows the information of the 18 web applications used as the benchmarks. These web applications have diverse functionalities and include the test set of NAVEX [16], the popular CMS applications listed by W3Techs [17], and a high-scoring CMS project on GitHub, etc. Thus, the selected web applications can provide comprehensive evaluation results.

2) Environment: We conducted the experiments on a system running Ubuntu 18.04 with an Intel Core i7-9750 (2.60 GHz) CPU with 32 GB of RAM. The target web applications are installed in an Ubuntu 16.04 system virtual machine with an Intel Core i7-9750 (2.60 GHz) CPU and 4 GB of RAM. We installed Apache 2.4 and PHP 7.0.3 on the target system. To ensure a consistent test environment for each tool, we take a snapshot of each web application's virtual machine and restore the virtual machine snapshot after each tool test is complete. We compared URadar with the state-of-the-art dynamic UFU vulnerability detection tool FUSE and the static program analysis tool RIPS. Since the static program analysis tools UChecker [7] and UFuzzer [8] are not open source, we could not conduct experimental comparisons with them. Note that all compared tools were run in the same environment for fairness.

B. Discovering UFU Vulnerabilities

Table II presents the UFU vulnerabilities discovered by URadar, FUSE, and RIPS in the 18 web applications. These experimental results provide the answer to RQ1. In dynamic testing, we tested for UFU vulnerabilities using four uploaded file types: PHP, HTML, XHTML, and JS. The *#Upload Interfaces* column in Table II indicates the number of file upload interfaces identified by URadar through the state-aware page traversal method. Among the 18 web applications, four had only one identified file upload interface. We manually analyzed the interfaces of these four web application pages and found that they each indeed had only one file upload interface among their web pages. For Drupal and Backdrop, URadar identified 14 and 15 file upload interfaces, respectively. The *CE* columns indicate the numbers of unique HTTP request packets that can be used to execute the code in *code executable* files through uploading the file of the PHP, HTML, and XHTML types. A positive number in one of these columns indicates that the corresponding application has UEFU vulnerabilities. URadar found 94 CEs in the 18 web applications, 28 of which were not found by FUSE. These 28 CEs were distributed among 7 web applications, including the well-known application WordPress, XE, and GetSimpleCMS. It must be emphasized that we further found that some of these 28 CEs still exist in the latest versions of these web applications, which have been tested for multiple years. Moreover, URadar detected UEFU vulnerabilities in 16 of the 18 web applications, while RIPS was able to detect UEFU vulnerabilities in only 2 web applications. Table II shows the evaluation results of RIPS. The results illustrate that URadar outperforms both FUSE and RIPS in detecting CE accurately and effectively.

The *PCE* columns indicate the numbers of unique request packets that can be used to upload *potentially code executable* files. In general, more PCE means that a web application has more potential flaws. Even if a file containing malicious code is successfully uploaded, if it cannot be parsed, execution of the code contained in the file cannot be triggered. However, if there is any file inclusion vulnerability [18] in the web application, which can include the PCE files, then the code in the file can be successfully executed. For PCE, URadar focuses not on quantity but on accuracy and efficiency. For example, consider a web application that does not restrict any non-executable files from being uploaded. All mutation chains that combine characteristics of non-executable files can be uploaded successfully and judged as PCE, which will cause resource waste during testing. Overall, in 16 of the 18 target web applications, URadar either outperforms FUSE or at least achieves the same performance. URadar found 547 PCEs in the 18 web applications, 228 more than FUSE, and RIPS

TABLE II
THE PERFORMANCE OF URADAR IN DISCOVERING UFU VULNERABILITIES

Application	URadar								FUSE							RIPS
	# Upload Interfaces	CE			PCE		Total	Code Executed	CE			PCE		Total	Code Executed	CE
		PHP	HTML	XHTML	PHP	JS			PHP	HTML	XHTML	PHP	JS			
Elgg	3	1	1	1	**1**	1	5	11,546(11.95%)	1	1	1	0	1	4	10,098(10.45%)	0(1)
Subrion	8	11	1	1	**74**	1	88	15,233(4.77%)	1	1	1	48	1	52	5,524(1.73%)	0(5)
Monstra	1	3	12	1	14	14	44	1,816(1.70%)	2	12	1	15	14	44	1,816(1.70%)	1(2)
CMSMadeSimple	3	2	1	1	14	1	19	8,913(2.87%)	2	1	1	14	1	19	6,534(2.11%)	0(2)
ClipperCMS	1	0	1	1	7	1	10	5,364(8.86%)	0	1	1	7	1	10	5,364(8.86%)	0(4)
Bludit	2	**1**	**5**	**1**	3	**2**	12	2,339(5.27%)	0	1	0	3	1	5	1,653(3.73%)	0(7)
MyBB	3	0	1	0	**37**	4	42	4,676(1.97%)	0	1	0	33	4	38	4,209(1.77%)	0(0)
Backdrop	15	0	**1**	**1**	**82**	**7**	91	44,410(60.23%)	0	0	0	34	1	35	16,010(21.71%)	0(0)
Joomla	2	0	0	0	6	2	8	16,020(47.19%)	0	0	0	**28**	2	30	9,171(27.01%)	0(0)
XE	3	**8**	**5**	**5**	**49**	**4**	71	16,404(11.30%)	0	2	2	1	1	6	16,007(11.03%)	0(0)
Wordpress	3	**2**	**3**	**6**	22	**4**	37	17,031(1.36%)	0	1	4	**43**	2	50	15,103(1.21%)	0(3)
phpBB	2	0	0	0	**22**	**4**	26	7,160(3.70%)	0	0	0	9	2	11	5,400(2.79%)	0(0)
Drupal	14	1	0	0	**124**	**7**	132	34,602(11.30%)	0	0	0	18	0	18	19,306(6.30%)	0(0)
CMSimple	2	0	**2**	**1**	**10**	**14**	27	6,150(11.75%)	0	1	0	5	3	9	3,131(5.98%)	0(1)
Concrete	2	0	3	2	6	4	15	20,897(64.04%)	0	3	2	6	4	15	18,471(56.60%)	0(0)
GetSimpleCMS	1	**2**	9	1	13	12	37	1,302(2.84%)	0	9	1	**15**	12	37	1,302(2.84%)	1(2)
SilverStripe	1	0	2	2	8	5	17	10,878(5.80%)	0	2	2	8	5	17	10,878(5.80%)	0(0)
Symphony	2	**2**	1	1	**16**	1	21	3,789(2.93%)	1	1	1	14	1	18	3,677(2.85%)	0(0)

Bold text indicates that the method used to obtain the corresponding result detects the highest number of CE or PCE. The numbers in parentheses in the RIPS column represent the numbers of vulnerabilities reported by RIPS, while the numbers outside the parentheses indicate the actual numbers of vulnerabilities after manual verification.

cannot find PCE. These 228 new found PCEs are mainly distributed among 10 tested web applications.

The *LoC Executed* columns indicate the number of lines of code executed by URadar and FUSE during file uploading testing. We can observe URadar outperforms FUSE in terms of code coverage. For 18 web applications, URadar covers 12,316 lines of code in average, while FUSE covers 8,536 lines of code in average. Among these web applications, URadar achieves better performance than FUSE in 14 web applications. For the other 4 web applications, URadar covers the same coverage as FUSE does. The main reason is that these web applications only have one file upload interface, making the related code very limited. On the other hand, the experimental results demonstrate the importance of identifying more interfaces, as it can help cover more code, bringing more comprehensive testing.

From the experimental results, we can also see that for Joomla and WordPress, URadar found fewer PCEs than did FUSE. We manually analyzed the experimental results of these two web applications. For Joomla, the results are as follows: once URadar found M05 combined with PNG, JPG, GIF, and PDF in M04 that could be successfully uploaded, URadar no longer combined the above mutation chain with other mutation operations to generate new test cases. In contrast, the 28 additional PCEs found by FUSE were formed by combining the above mutation chain with other mutation operations. This occurred because once URadar identifies that it can successfully upload a combined mutation chain that affects properties of a single non-executable file type, it will not generate a combined mutation chain that involves attributes from different non-executable file types. The same situation occurred for WordPress.

In addition, we compare UFuzzer and URadar. Since UFuzzer is not open source, we choose to make the comparisons based on the experimental data provided in the UFuzzer paper. Considering that URadar is implemented based on FUSE, its capability can cover what FUSE has. To avoid redundant experiments, we exclude systems in which FUSE had already detected file upload vulnerabilities in the UFuzzer experiments. Finally, we select 11 systems for comparison with UFuzzer. Note that, PDW Media File Browser version 1.1 is not available for download. In the UFuzzer paper, the authors only evaluated whether UFuzzer can find vulnerabilities in these systems. Thus, we follow this same approach for consistency, and the results are shown in Table III. We can see that URadar is capable of detecting UFU and UEFU vulnerabilities present in these systems.

1) UEFU Vulnerabilities: Although UEFU vulnerabilities are a subclass of UFU vulnerabilities, UEFU vulnerabilities pose greater security risks to a system. Therefore, we focus more on the discovery of UEFU vulnerabilities. URadar found 26 UEFU vulnerabilities in the 18 web applications, while FUSE found 14 and RIPS found 2. Among the UEFU vulnerabilities discovered by URadar, eight are new. We have reported the newly discovered UEFU vulnerabilities to the corresponding vendors and received positive responses.

TABLE III
COMPARISON BETWEEN URADAR AND UFUZZER

System	Ufuzzer	URadar
Adblock Blocker 0.0.1	✓	✓
Audio Record 1.0	✓	✓
Estatik 2.2.5	✓	✓
File Provider 1.2.3	✓	✓
Image Gallery with Slideshow 1.5.2	✓	✓
Open Flash Chart Core 0.4	✓	✓
PDW Media File Browser 1.1	✓	N/A*
Ip Blocker Lite 10.2	✓	✓
Uploadify 1.0.0	✓	✓
WooCommerce Custom Profile Picture 1.0	✓	✓
WP Demo Buddy 1.0.2	✓	✓

The asterisk (*) indicates that the system is no longer available for download, so we cannot conduct experimental evaluations on it.

Currently, we have obtained four new CVE IDs and two new CNNVD IDs, namely, CVE-2021-44911, CVE-2021-44912, CVE-2021-28976, CVE-2021-28977, CNNVD-202112-2684, and CNNVD-202112-2685. Based on our analysis, the reason why URadar was able to discover these new UEFU vulnerabilities is its ability to identify such vulnerabilities across different PHP versions. This feature enhances its capability for vulnerability detection.

2) False Positive/Negative: Regarding the false positive rate, since both URadar and FUSE employ dynamic testing to detect UFU and UEFU vulnerabilities, they will only report vulnerabilities that are triggered by the executed test cases. By manual analysis and verification, we find that URadar and FUSE do not produce false positives in these 18 web applications. This is also the advantage of dynamic testing in terms of accuracy in detecting vulnerabilities. Table II demonstrates that RIPS generates a substantial number of false positives. The number of vulnerabilities reported by RIPS is denoted within parentheses, while the actual number confirmed after manual analysis is displayed outside the parentheses. Specifically, RIPS reports a total of 27 CEs across 18 web applications. However, after manual analysis, 2 CEs are actually identified, and the false positive rate is 92.59%. We analyze the reasons for the high false positives of RIPS, which is due to the user-defined sanitization actions are frequently employed for security checking in the file upload process and RIPS's inability to account for user-defined sanitization actions during vulnerability modeling.

We evaluate the false negative rate of URadar, FUSE, and RIPS using all the detected vulnerabilities in each web application as a baseline. As shown in Table IV, there are a total of 26 UEFU vulnerabilities in these 18 web applications. URadar is able to detect all UEFU vulnerabilities in each tested web application. In comparison, FUSE detects 14 UEFU vulnerabilities, with a false negative rate of 46.15%, while RIPS detects 2 UEFU vulnerabilities, leading to a false negative rate of 92.31%.

3) Efficiency: To answer RQ2, we assessed the efficiency of URadar in terms of the number of packets sent and the execution time required to identify the same number of UFU vulnerabilities within the same file upload form of a web application. We compared these results with those obtained from the state-of-the-art dynamic testing tool FUSE. The web application's file upload form URL address is shown in Table V. The experimental results are shown in Table VI. The *File Upload Request Address* column refers to the request address of the file form in the target web application tested by URadar and FUSE. In Table VI, the *Request Packets* columns represent the total number of upload request packets sent by FUSE and URadar. The *Efficiency Improvement(Packets)* column indicates the percentage by which URadar reduces the number of packets sent compared to FUSE. The *Execution Time* columns represent the execution times of FUSE and URadar. The *Efficiency Improvement(Time)* indicates the efficiency improvement of URadar in execution time compared to FUSE. Based on these results, the efficiency of the discovery of UFU and UEFU vulnerabilities by URadar can be evaluated.

The experimental results show that when URadar and FUSE test for the same file upload form of the same web application, URadar outperforms FUSE in terms of efficiency. As shown in Fig. 6, among the experimental results for the 18 web applications, the test efficiency for 17 web applications is increased by more than 90%. For example, the numbers of upload request packets sent to Elgg, Bludit, XE, and Drupal are reduced from 127,160, 117,267, 105,757, and 120,849, respectively, to 3,873, 1,080, 6,287, and 1,236. At the same time, the execution times are reduced from 108m2s, 68m48s, 13m56s, and 265m25s to 6m3s, 34.5s, 2m35s, and 2m43s, respectively. Compared with FUSE, URadar achieves approximately 76.87% and 80.81% improvements in the number of HTTP request packets and execution time, respectively. The experimental results in terms of request packets and execution time are similar for these tested web applications.

We document the experimental results of the file type restriction inference method in Table VI. For the 18 web applications, the cross-location method sent an average of 296 HTTP request packets, and the average execution time was 15.54 s. Here, we analyze the experimental results for Subrion, which is the only test application for which the efficiency improvement was negative. Although the efficiency improvement was -231.67% and -100%, the number of HTTP request packets sent by URadar increased by only 139, and the execution time increased by only 7s. The main reason for the negative efficiency improvements is that the UFU and UEFU vulnerabilities in Subrion were discovered based on successfully uploading a single mutation operation. In this unique scenario, the file type restriction inference method in URadar required sending additional HTTP request packets to gather the corresponding information. This is evident from the fact that 159 of the HTTP requests sent by URadar to Subrion and 10 seconds of its execution time were due to the cross-location method. Subsequently, only 40 single mutation operation HTTP requests were sent by URadar, indicating that URadar eliminated 20 invalid test cases compared to FUSE, and the remaining execution time was only 4s.

The results in Fig. 6 show more intuitively that overall, the number of request packets sent by URadar to a target web

TABLE IV
THE FALSE NEGATIVE FOR URADAR, FUSE AND RIPS

Applications	Version	Total number of UEFU vulnerabilities	URadar		FUSE		RIPS	
			UEFU	False Negative	UEFU	False Negative	UEFU	False Negative
Elgg	2.3.10	1	1	0%	1	0%	0	100%
Subrion	4.2.1	2	2	0%	1	50%	0	100%
Monstra	3.0.4	2	2	0%	1	50%	1	50%
CMSMadeSimple	2.2.9.1	1	1	0%	1	0%	0	100%
ClipperCMS	1.3.3	1	1	0%	1	0%	0	100%
Bludit	3.8.1	2	2	0%	1	50%	0	100%
MyBB	1.8.19	1	1	0%	1	0%	0	100%
Backdrop	1.12.1	1	1	0%	0	100%	0	100%
Joomla	3.9.3	0	0	0%	0	0%	0	0%
XE	1.11.2	4	4	0%	1	75%	0	100%
Wordpress	5.0.3	3	3	0%	1	66.67%	0	100%
phpBB	3.2.5	0	0	0%	0	0%	0	0%
Drupal	8.6.9	1	1	0%	0	100%	0	100%
CMSimple	4.7.7	1	1	0%	1	0%	0	100%
Concrete	8.4.4	1	1	0%	1	0%	0	100%
GetSimpleCMS	3.3.15	2	2	0%	1	50%	1	50%
SilverStripe	4.3.0	1	1	0%	1	0%	0	100%
Symphony	2.7.7	2	2	0%	1	50%	0	100%
Total		26	26	0%	14	46.15%	2	92.31%

TABLE V
THE FILE UPLOAD ADDRESS FOR EFFICIENCY COMPARISON BETWEEN FUSE AND URADAR

Application	File Upload Request Address
Elgg	/action/avatra/upload
Subrion	/panel/uploads/read.json
Monstra	/admin/index.php?id=filemanager
CMSMadeSimple	/admin/moduleinterface.php
ClipperCMS	/manager/index.php?a=31
Bludit	/admin/ajax/upload-images
MyBB	/newthread.php?fid=2&processed=1
Backdrop	/node/add/post
Joomla	/administrator/index.php?option=com_media&folder=test
XE	/index.php?module=file&act=procFileUpload&mid=board
WordPress	/wp-admin/async-upload.php
phpBB	/posting.php?mode=post&f=2
Drupal	/node/add/article
Concrete	/index.php/ccm/system/file/upload
GetSimpleCMS	/admin/upload.php?path=
SilverStripe	/admin/assets/api/createFile
WeBid	/admin/logo_upload.php
Symphony	/symphony/publish/articles/new/

application is significantly reduced compared to the number sent by FUSE.

To answer RQ3, we used the selected 18 web applications to verify the effectiveness of the file type restriction inference method (FTRI/URadar) and the invalid mutation combination filtration method (IMCF/URadar). The experimental results are shown in Fig. 7. The different methods can be ranked as follows in terms of the number of HTTP request packets shown: URadar $<$ IMCF/URadar $\approx$ FTRI/URadar $\ll$ FUSE. The number of upload HTTP request packets sent by IMCF/URadar is the same as or lower than that of FUSE when the seed file itself or the outcome of one mutation operation on the seed file can be uploaded successfully, as in the case of Subrion. In other cases, IMCF/URadar is significantly more efficient than FUSE.

The experimental results in Fig. 7 fully illustrate the effectiveness of the proposed combined file type restriction inference and invalid mutation filtering methods. These methods can greatly improve the effectiveness of test cases and reduce the generation of invalid mutation cases. The combination of these two methods can exert a more significant effect.

Moreover, we evaluate the performance of URadar on web applications with multiple versions and compare it with FUSE. The experimental results are shown in Table VII. On the three web applications with different versions, URadar has better performance than FUSE in both vulnerability detection and code coverage. In terms of CE, URadar finds 93 CEs among these nine targets, while FUSE finds 60 CEs. In terms of PCE, URadar detects 361 PCEs, compared to FUSE's 331 PCEs. Regarding of code coverage, URadar outperforms FUSE in different versions of Symphony and Bludit. The average number of executed code lines by URadar is 3,764 and 2,447, compared to FUSE's averages of 3,643 and 1,728, respectively. In the multiple versions of GetSimpleCMS, URadar's code coverage performance is similar to FUSE. The main reason is that GetSimpleCMS only has one file upload interface, making the related code very limited.

C. Case Studies

Listing 5 presents the security detection code for file upload operations in GetSimpleCMS 3.3.16 [19]. When a user attempts to upload a file, GetSimpleCMS assesses its admissibility. It calls the `validate_safe_file` function to cross-check the file's MIME type and extension against predefined blacklists, the blacklists as shown in Listing 6. If neither the MIME type nor the file extension is blacklisted, the function returns `True`, indicating the file is safe for upload. The `file_mime_type` function uses the built-in

TABLE VI
COMPARISON OF TEST EFFICIENCY BETWEEN URADAR AND FUSE FOR THE SAME UPLOAD FORM

Application	Request Packets FUSE	Request Packets (URadar)	Efficiency improvement (Packet)	Execution Time (FUSE)	Execution Time (URadar)	Efficiency Improvement (Time)
Elgg	127,160	3,688	97.10%	108m2s	5m40s	93.21%
Subrion	60	40	33.33%	7s	4s	42.86%
Monstra	16,982	1,100	93.52%	12m35s	2m18s	89.34%
CMSMadeSimple	24,986	1,934	92.26%	24m11s	1m55s	92.07%
ClipperCMS	63,259	3,046	95.18%	3m5s	30s	83.78%
Bludit	117,267	1,066	99.09%	68m48s	34s	99.18%
MyBB	12,142	474	96.10%	2m40s	7s	95.63%
Backdrop	26,930	2,395	91.11%	15m29s	1m36s	89.67%
Joomla	121,117	21,092	82.59%	42m48s	8m42s	79.67%
XE	105,757	6,189	94.15%	13m56s	2m30s	82.05%
Wordpress	98,730	6,027	93.90%	16m22s	2m41s	83.60%
phpBB	119,796	4,428	96.30%	17m51s	35s	96.73%
Drupal	120,849	1,115	99.08%	265m25s	2m28s	99.07%
CMSsimple	102,168	4,660	95.44%	24m35s	2m47s	88.68%
Concrete	96,638	1,872	98.06%	24m35s	49s	97.80%
GetSimpleCMS	52,564	2,861	94.56%	37m10s	2m4s	97.73%
SilverStripe	87,312	6,421	92.65%	91m19s	5m25s	93.15%
Symphony	24,980	1,059	95.76%	4m54s	16s	94.56%
Average	73,261	3,859	91.12%	2791.1s	317.5s	88.82%

The data in parentheses in the relevant columns for URadar represent the number of HTTP request packets sent in the cross-location method and the execution time of the cross-location method.

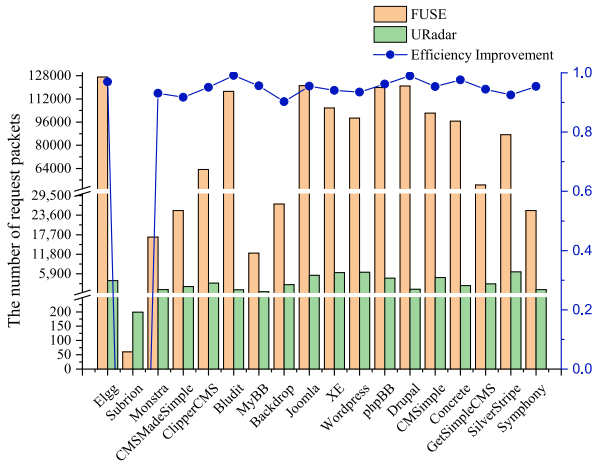


Fig. 6. The performance of URadar compared to FUSE.

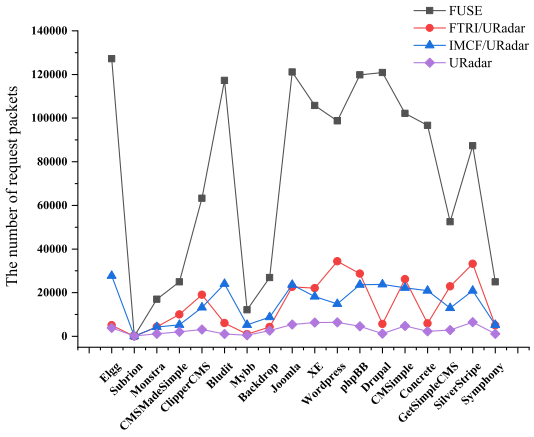


Fig. 7. The performance of FTRI/URadar, IMCF/URadar and URadar compared to FUSE.

PHP function `finfo_file` [20] to obtain the file's MIME type. The `finfo_file` function obtains the MIME type by identifying the file header information. URadar can bypass it by adding the file header information for a file type that is not on the blacklist when uploading the file, Furthermore, URadar can find that the blacklist does not contain two types of executable extensions: ".php7" and ".phar". Therefore, by forging the file header information and exploiting the incomplete file extension blacklist, URadar can easily bypass the security check mechanism of the `validate_safe_file` function and successfully upload an executable file. Consequently, this file upload form has a UEFU vulnerability. Notably, the PHP version used in our experimental environment is 7.0.3, which does not support the parsing and execution of ".php7" and ".phar" files, URadar has the ability to discover UFU vulnerabilities across PHP versions and can quickly discover this vulnerability. These vulnerabilities have obtained CVE-2021-28976 and CVE-2021-28977. Attackers can exploit these vulnerabilities to get the control permissions of the web application server directly, threatening system security severely.

TABLE VII
URADAR'S PERFORMANCE IN MULTIPLE VERSIONS OF WEB APPLICATIONS

Name	Version	URadar							FUSE						
		CE			PCE		Total	LoC Executed	CE			PCE		Total	LoC Executed
		PHP	HTML	XHTML	PHP	JS			PHP	HTML	XHTML	PHP	JS		
GetSimpleCMS	**3.3.14**	2	9	1	13	12	**37**	1,300(2.84%)	0	9	1	15	12	37	1,300(2.84%)
	3.3.15	2	9	1	13	12	37	1,302(2.84%)	0	9	1	15	12	37	1,302(2.84%)
	3.3.16	13	27	3	197	12	252	1,367(2.96%)	0	27	3	210	12	252	1364(2.96%)
Symphony	**2.7.5**	2	1	1	16	1	21	3,698(2.87%)	1	1	1	14	1	18	3,550(2.75%)
	2.7.7	2	1	1	16	1	21	3,789(2.93%)	1	1	1	14	1	18	3,677(2.84%)
	2.7.10	2	1	1	16	1	21	3,806(2.95%)	1	1	1	14	1	18	3701(2.86%)
Bludit	**3.8.1**	1	5	1	3	2	12	2,339(5.27%)	0	1	0	3	1	5	1,653(3.73%)
	3.11.0a	0	7	0	23	0	30	2,467(4.09%)	0	0	0	4	0	4	1738(2.88%)
	3.15.0	0	0	0	23	0	23	2,534(2.97%)	0	0	0	4	0	4	1793(2.10%)

```php
<?php
...
function validate_safe_file($file,$name,$mime = null){
    ...
    if(!$mime)$mime = file_mime_type($file);
    ...
    if ($mime && in_arrayi($mime,$mime_type_blacklist)){
        return false;
    }elseif(in_arrayi($file_extension,$file_ext_blacklist)){
        return false;
    }else{
        return true;
    }
}
function file_mime_type($file){
    ...
    if(function_exists('finfo_open')){
        $finfo = finfo_open(FILEINFO_MIME_TYPE);
        $mimetype = finfo_file($finfo,$file);
        finfo_close($finfo);
    }elseif(function_exists('mime_content_type')){
        $mimetype = mime_content_type($file);
    }else {
        return false;
        exit;
    }
    return $mimetype;
}
...
?>
```

Listing. 5. The code of file security detection in GetSimpleCMS.

```php
<?php
...
$mime_type_blacklist = array(
    'text/html','text/javascript','text/x-javascript',
    'application/x-shellscript','application/x-php',
    'text/x-php','text/x-python', 'text/x-perl',
    'text/x-bash', 'text/x-sh', 'text/x-csh',
    'text/scriptlet', 'application/x-msdownload',
    'application/x-msmetafile','application/x-opc+zip');
$file_ext_blacklist = array(
    'html', 'htm', 'js', 'jsb', 'mhtml', 'mht','php', 'pht',
    'phtm', 'phtml', 'php3', 'php4', 'php5', 'ph3', 'ph4',
    'ph5', 'phps','shtml', 'jhtml', 'pl', 'py', 'cgi', 'sh',
    'ksh', 'bsh', 'c', 'htaccess', 'htpasswd','exe', 'scr',
    'dll', 'msi', 'vbs', 'bat', 'com', 'pif', 'cmd', 'vxd',
    'cpl'
    );
...
?>
```

Listing. 6. The code of the blacklists for file types and extensions in GetSimpleCMS.

V. DISCUSSION

A. Limitations

Although URadar achieves good performance in discovering UFU vulnerabilities, it still has limitations that needs to be improved in the future. For instance, URadar is not good at finding UFU vulnerabilities caused by conditional contention or unknown errors in check functions. Additionally, complex logic errors can contribute to certain UFU vulnerabilities, necessitating the development of more advanced methods to detect them.

B. Possible Defense

The essence of UFU vulnerabilities resides in the flaws of web servers' security checks. Most existing security checks are based on simple rules, e.g., a blacklist of file extension types, which may be unable to defend against various file upload attacks. We present three possible defense approaches based on an in-depth analysis of our discovered UFU vulnerabilities. (1) A combination of a file extension whitelist and file content security detection can be used to defend against UFU vulnerabilities. (2) Cloud storage of files can be used for vulnerability defense. (3) The file resolution rules of the directory where the uploaded files are located can be strictly limited.

C. Future Work

1) UFU Vulnerabilities in New Web Scenarios: With the emergence and rise of new web scenarios, such as cloud storage based web applications, it is important to detect UFU vulnerabilities in these new scenarios. Taking SaaS (Software as a Service) as an example, detecting UFU vulnerabilities in SaaS based web applications may face the following challenges. (1) It is not trivial to understand the complex interaction mechanism between different roles and determine whether it contains vulnerabilities. Many interactions are invisible to a regular web user. For instance, in practice, a user is not able to access the communication process between a

website and the cloud storage. (2) It is not trivial to determine whether the concrete implementations or deployments of an interaction mechanism are vulnerable. SaaS usually includes multiple microservices, and their deployment methods may vary, with each service possibly having its own security configuration and policies. In light of these challenges, we plan to delve deeper into research on emerging technologies in the future.

2) Universality of URadar: While our paper focuses on the exploration of UFU and UEFU vulnerabilities in PHP, the core design, including file upload interface identification, file type restriction inference, and invalid mutation combination filtration, are all built upon universally applicable web security principles and techniques, not just attributes specific to PHP. In future work, we will explore how to make URadar's seed and mutation operations adaptable to different web development languages, enhancing the versatility of URadar.

VI. Related Work

Research on vulnerability discovery in web applications is divided into two main approaches: static analysis and dynamic testing. Static analysis is based on the source code of a web application and is conducted through data flow analysis [21], symbolic execution [22], [23] or taint propagation [24]. Dynamic testing starts from the perspective of web application services. By exploring parameter relationships, it sends test cases of the HTTP protocol to the data interface of a web application service to determine whether a corresponding error is triggered for vulnerability discovery.

1) Static Analysis: Static analysis has been widely used in web application vulnerability discovery, where representative ones include [6], [7], [8], [16], [25], [26], [27], [28], [29], [30], [31], [32], [24], [33]. Specifically, [26], [27], [28], [30] utilize data flow analysis in static analysis to explore SQL injection and cross-site scripting vulnerabilities. As web applications become more feature-rich and business logic becomes increasingly complex, coarse-grained static analysis methods are beginning to show drawbacks such as high false positive rates. Alhuzali et al. [16] proposed NAVEX, which combines static and dynamic analysis methods and can maximize the coverage of the paths in an application. NAVEX conducts a more fine-grained static analysis of the code and uses dynamic testing to improve the accuracy of vulnerability detection. While these methods have achieved certain results and shown great promise in detecting vulnerabilities in web application source code, they focus on conventional ones other than unrestricted file upload vulnerabilities.

A few existing studies [5], [6], [7], [8] have the ability to detect unrestricted file upload vulnerabilities based on static analysis methods. RIPS [5], [6] adopts inter-procedural data flow analysis and taint analysis methods. Although RIPS's taint analysis process considers the source of the uploaded file, it does not comprehensively model attributes such as the file's content, type, size, and filename, leading to false positives. UChecker [7] is based on symbolic execution and taint analysis technology. The authors designated the file upload function as a sink point, used SMT for path modeling, and calculated the satisfiability of the constraints on each path to determine whether a UFU vulnerability exists. However, the semantic gaps between PHP and the solver language introduce intrinsic challenges for constraint modeling. Later, Huang et al. proposed UFuzzer [8] based on UChecker, which is more advanced than UChecker. However, due to the dynamic nature of the PHP language, it is challenging to discover all possible paths in source code through static analysis alone.

2) Dynamic Testing: Dynamic testing has also begun to be used in web application systems to explore various types of vulnerabilities, where representative studies include [34], [35], [36], [37], [38], [39], [40], [41], [42], [43]. Wassermann et al. [34] proposed an automated input test case generation algorithm that uses string manipulation to model and collect constraint relationships between parameters on a path to generate test cases to explore SQL injection vulnerabilities in web application systems. Eriksson et al. [39] proposed a dynamic crawler called Black Window based on state dependence. Black Window models links, forms, events, and their interactions, covering the attack surface of a web application as much as possible, with a focus on testing and discovering XSS vulnerabilities in web application systems. Inspired by these works, we conducted a more fine-grained analysis of web pages in order to discover the file upload interface in web applications as comprehensively as possible. Although these studies did not focus on unrestricted file upload vulnerabilities, we were inspired by them to conduct a more fine-grained analysis of web pages, aiming to comprehensively identify file upload interfaces in web applications.

FUSE [2] is the state-of-the-art tool for file upload vulnerability detection based on dynamic analysis. FUSE embedded with 13 carefully designed mutation operations and their combinations. Inspired by their research, we focused on reducing human effort in the testing process and using effective code information to guide test case generation, aiming to enhance the efficiency and accuracy of vulnerability detection.

VII. Conclusion

In this paper, we propose URadar, an adaptive dynamic testing method for discovering UFU vulnerabilities. By means of three core designs, namely, *file upload interface identification*, *file type restriction inference*, and *invalid mutation combination filtration*, URadar is able to reduce the number of generated invalid test cases, thus significantly improving its efficiency in discovering UFU vulnerabilities. In addition, URadar has the ability to discover UFU vulnerabilities across PHP versions, eliminating the need for different versions of the PHP parsing environment, thereby improving accuracy and reducing false positives. We have evaluated the performance of URadar in discovering UFU vulnerabilities in 18 popular web applications. The experimental results demonstrate that URadar shows significant advantages in terms of both effectiveness and efficiency. URadar found 26 UEFU vulnerabilities in the 18 popular web applications, whereas FUSE found 14, and RIPS found 2. Among the UEFU vulnerabilities discovered by URadar, eight are new. We have reported the newly discovered UEFU vulnerabilities to the corresponding vendors, and six have been assigned new CVE/CNNVD IDs. In the future, we will conduct research seeking to improve the ability

of URadar to detect more complicated vulnerabilities in web applications.

ACKNOWLEDGMENT

The authors would like to thank the anonymous reviewers for their detailed comments, which helped to improve the quality of the article.

DECLARATIONS

Conflict of Interests: We declare that we have no conflicts of interest.

REFERENCES

[1] Contributors. (2018). *Unrestricted File Upload*. [Online]. Available: https://www.owasp.org/index.php/Unrestricted_File_Upload
[2] T. Lee, S. Wi, S. Lee, and S. Son, "FUSE: Finding file upload bugs via penetration testing," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2020, pp. 1–17.
[3] (2021). *Broken Access Control*. [Online]. Available: https://www.owasp.org/index.php/Broken_ Access_Control
[4] *About the Owasp Foundation*. Accessed: 2023. [Online]. Available: https://owasp.org/about/
[5] J. Dahse and J. Schwenk, "RIPS-A static source code analyser for vulnerabilities in PHP scripts," in *Seminar Work (Seminer Çalismasi). Horst Görtz Institute Ruhr-University Bochum*. Citeseer, 2010.
[6] J. Dahse and T. Holz, "Simulation of built-in PHP features for precise static code analysis," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2014, pp. 23–26.
[7] J. Huang, Y. Li, J. Zhang, and R. Dai, "UChecker: Automatically detecting PHP-based unrestricted file upload vulnerabilities," in *Proc. 49th Annu. IEEE/IFIP Int. Conf. Dependable Syst. Netw. (DSN)*, Jun. 2019, pp. 581–592.
[8] J. Huang, J. Zhang, J. Liu, C. Li, and R. Dai, "UFuzzer: Lightweight detection of PHP-based unrestricted file upload vulnerabilities via static-fuzzing co-analysis," in *Proc. 24th Int. Symp. Res. Attacks, Intrusions Defenses*, Oct. 2021, pp. 78–90.
[9] H. Su et al., "A sanitizer-centric analysis to detect cross-site scripting in PHP programs," in *Proc. IEEE 33rd Int. Symp. Softw. Rel. Eng. (ISSRE)*, Oct. 2022, pp. 355–365.
[10] PHP. (2021). *PHP: PHP 5.6.0 Release Announcement*. [Online]. Available: https://www.php.net/releases/5_6_0.php
[11] PHP. (2021). *PHP: PHP 7.0.0 release announcement*. [Online]. Available: https://www.php.net/releases/7_0_0.php
[12] PHP. (2021). *PHP: PHP 7.2.0 Release Announcement*. [Online]. Available: https://www.php.net/releases/7_2_0.php
[13] WordPress. (2021). *Blog Tool, Publishing Platform, and CMS*. [Online]. Available: https://wordpress.org/
[14] Joomla. (2021). *Joomla Content Management System (CMS)*. [Online]. Available: https://www.joomla.org/
[15] E. Rescorla, "Multipurpose internet mail extensions (MIME) part one: Format of internet message bodies," *RFC*, vol. 47, no. 5, pp. 9–11, 1996.
[16] A. Alhuzali, R. Gjomemo, B. Eshete, and V. Venkatakrishnan, "NAVEX: Precise and scalable exploit generation for dynamic web applications," in *Proc. USENIX Secur. Symp.*, 2018, pp. 377–392.
[17] W3Techs. (2021). *Usage Statistics and Market Share of Content Management Systems*. [Online]. Available: https://w3techs.com/technologies/overview/content_management
[18] OWASP. (2021). *PHP File Inclusion | Owasp*. [Online]. Available: https://owasp.org/www-community/vulnerabilities/PHP_File_Inclusion
[19] GetSimpleCMS. (2021). *Getsimple CMS*. [Online]. Available: http://get-simple.info/download
[20] PHP. (2021). *Finfo-File—Manual*. [Online]. Available: https://www.php.net/manual/en/function.finfo-file.php
[21] H. Grent, A. Akimov, and M. Aniche, "Automatically identifying parameter constraints in complex web APIs: A case study at adyen," in *Proc. IEEE/ACM 43rd Int. Conf. Softw. Eng., Softw. Eng. Pract. (ICSE-SEIP)*, May 2021, pp. 71–80.
[22] P. Li, W. Meng, K. Lu, and C. Luo, "On the feasibility of automated built-in function modeling for PHP symbolic execution," in *Proc. Web Conf.*, Apr. 2021, pp. 58–69.
[23] R. Baldoni, E. Coppa, D. C. D'elia, C. Demetrescu, and I. Finocchi, "A survey of symbolic execution techniques," *ACM Comput. Surv.*, vol. 51, no. 3, pp. 1–39, May 2019.
[24] D. Balzarotti et al., "Saner: Composing static and dynamic analysis to validate sanitization in web applications," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2008, pp. 387–401.
[25] T. Jensen, H. Pedersen, M. C. Olesen, and R. R. Hansen, "THAPS: Automated vulnerability scanning of PHP applications," in *Proc. Nordic Conf. Secure IT Syst.* Cham, Switzerland: Springer, 2012, pp. 31–46.
[26] N. Jovanovic, C. Kruegel, and E. Kirda, "Pixy: A static analysis tool for detecting web application vulnerabilities," in *Proc. IEEE Symp. Secur. Privacy*, 2006, pp. 1–6.
[27] G. Wassermann and Z. Su, "Sound and precise analysis of web applications for injection vulnerabilities," in *Proc. 28th ACM SIGPLAN Conf. Program. Lang. Design Implement.*, Jun. 2007, pp. 32–41.
[28] Y. Xie and A. Aiken, "Static detection of security vulnerabilities in scripting languages," in *Proc. USENIX Secur. Symp.*, vol. 15, 2006, pp. 179–192.
[29] S. Son and V. Shmatikov, "SAFERPHP: Finding semantic vulnerabilities in PHP applications," in *Proc. ACM SIGPLAN 6th Workshop Program. Lang. Anal. Secur.*, Jun. 2011, pp. 1–13.
[30] G. Wassermann and Z. Su, "Static detection of cross-site scripting vulnerabilities," in *Proc. 13th Int. Conf. Softw. Eng.*, 2008, pp. 171–180.
[31] J. Dahse and T. Holz, "Static detection of second-order vulnerabilities in web applications," in *Proc. USENIX Secur. Symp.*, 2014, pp. 989–1003.
[32] M. Shcherbakov and M. Balliu, "SerialDetector: Principled and practical exploration of object injection vulnerabilities for the web," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2021, pp. 1–18.
[33] A. Doupé, B. Boe, C. Kruegel, and G. Vigna, "Fear the EAR: Discovering and mitigating execution after redirect vulnerabilities," in *Proc. 18th ACM Conf. Comput. Commun. Secur.*, Oct. 2011, pp. 251–262.
[34] G. Wassermann, D. Yu, A. Chander, D. Dhurjati, H. Inamura, and Z. Su, "Dynamic test input generation for web applications," in *Proc. Int. Symp. Softw. Test. Anal.*, Jul. 2008, pp. 249–260.
[35] S. Artzi et al., "Finding bugs in dynamic web applications," in *Proc. Int. Symp. Softw. Test. Anal.*, Jul. 2008, pp. 261–272.
[36] A. Petukhov and D. Kozlov, "Detecting security vulnerabilities in web applications using dynamic analysis with penetration testing," Comput. Syst. Lab, Dept. Comput. Sci., Moscow State Univ., 2008, pp. 1–120.
[37] A. Doupe, M. Cova, and G. Vigna, "Why Johnny can't pentest: An analysis of black-box web vulnerability scanners," in *Proc. Int. Conf. Detection Intrusions Malware, Vulnerability Assessment*. Cham, Switzerland: Springer, 2010, pp. 111–131.
[38] P. Saxena, S. Hanna, P. Poosankam, and D. Song, "FLAX: Systematic discovery of client-side validation vulnerabilities in rich web applications," in *Proc. Netw. Distrib. Syst. Secur. Symp. (NDSS)*, 2010.
[39] B. Eriksson, G. Pellegrino, and A. Sabelfeld, "Black widow: Blackbox data-driven web scanning," in *Proc. IEEE Symp. Security Privacy*, May 2021, pp. 1125–1142.
[40] W. Tian, J.-F. Yang, J. Xu, and G.-N. Si, "Attack model based penetration test for SQL injection vulnerability," in *Proc. IEEE 36th Annu. Comput. Softw. Appl. Conf. Workshops*, Jul. 2012, pp. 589–594.
[41] S. Jan, C. D. Nguyen, and L. C. Briand, "Automated and effective testing of web services for XML injection attacks," in *Proc. 25th Int. Symp. Softw. Test. Anal.*, Jul. 2016, pp. 12–23.
[42] D. Appelt, C. D. Nguyen, L. C. Briand, and N. Alshahwan, "Automated testing for SQL injection vulnerabilities: An input mutation approach," in *Proc. Int. Symp. Softw. Test. Anal.*, Jul. 2014, pp. 259–269.
[43] K. Drakonakis, S. Ioannidis, and J. Polakis, "The cookie hunter: Automated black-box auditing for web authentication and authorization flaws," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Oct. 2020, pp. 1953–1970.

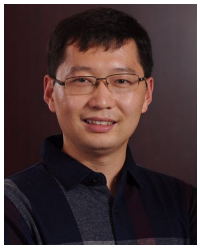
Yuanchao Chen received the bachelor's degree in network engineering from the National University of Defense Technology, Hefei, China, in 2019, where he is currently pursuing the Ph.D. degree. His research interests include web application security, vulnerability discovery, and cloud security.

Yuwei Li received the Ph.D. degree in computer science and technology from Zhejiang University, Hangzhou, China, in 2021. She is currently a Lecturer with the College of Electronic Engineering, National University of Defense Technology. Her research interests include system and software security, fuzz testing, and vulnerability detection.

Zulie Pan received the Ph.D. degree from the Electronic Engineering Institute, Hefei, China. He is currently a Professor with the College of Electronic Engineering, National University of Defense Technology. His research interests include vulnerability discovery, network security, and computer science.

Yuliang Lu was born in China in 1964. He received the B.Sc. degree (Hons.) in computer application in China in 1985 and the M.Sc. degree in computer application from Southeast University in 1988. He is currently a Professor with the National University of Defense Technology, Hefei, China. His research interests are computer application and information processing.

Juxing Chen received the bachelor's degree from the School of Computer Science and Artificial Intelligence, Wuhan University of Technology, Hubei, China. She is currently pursuing the master's degree with the College of Electronic Engineering, National University of Defense Technology, Anhui, China. Her research interests include web application security and web vulnerability discovery.



Shouling Ji (Member, IEEE) received the Ph.D. degree in electrical and computer engineering from the Georgia Institute of Technology and the Ph.D. degree in computer science from Georgia State University. He is currently a ZJU 100-Young Professor with the College of Computer Science and Technology, Zhejiang University, and a Research Faculty Member of the School of Electrical and Computer Engineering, Georgia Institute of Technology. His current research interests include AI security, data-driven security, privacy, and data analytics. He is a member of ACM and was the Membership Chair of the IEEE Student Branch with Georgia State University (2012–2013).