electronics

Article

APIMiner: Identifying Web Application APIs Based on Web Page States Similarity Analysis

Yuanchao Chen [1], Yuliang Lu [1,*,†], Zulie Pan [1,*,†], Juxing Chen [1], Fan Shi [1], Yang Li [1] and Yonghui Jiang [2]

1 College of Electronic Engineering, National University of Defense Technology, Heifei 230037, China; chenyuanchao@nudt.edu.cn (Y.C.); liyanghf@nudt.edu.cn (Y.L.)
2 Anhui Shenwu Information Technology Co., Ltd., Heifei 241000, China; i286@foxmail.com
* Correspondence: luyuliang@nudt.edu.cn (Y.L.); panzulie17@nudt.edu.cn (Z.P.)
† These authors contributed equally to this work.

Abstract: Modern web applications offer various APIs for data interaction. However, as the number of these APIs increases, so does the potential for security threats. Essentially, more APIs in an application can lead to more detectable vulnerabilities. Thus, it is crucial to identify APIs as comprehensively as possible in web applications. However, this task faces challenges due to the increasing complexity of web development techniques and the abundance of similar web pages. In this paper, we propose APIMiner, a framework for identifying APIs in web applications by dynamically traversing web pages based on web page state similarity analysis. APIMiner first builds a web page model based on the HTML elements of the current web page. APIMiner then uses this model to represent the state of the page. Then, APIMiner evaluates each element's similarity in the page model and determines the page state similarity based on these similarity values. From the different states of the page, APIMiner extracts the data interaction APIs on the page. We conduct extensive experiments to evaluate APIMiner's effectiveness. In the similarity analysis, our method surpasses state-of-the-art methods like NDD and mNDD in accurately distinguishing similar pages. We compare APIMiner with state-of-the-art tools (e.g., Enemy of the State, Crawlergo, and Wapiti3) for API identification. APIMiner excels in the number of identified APIs (average 1136) and code coverage (average 28,470). Relative to these tools, on average, APIMiner identifies 7.96 times more APIs and increases code coverage by 142.72%.

Keywords: web application; web API; state aware; similarity analysis

check for updates

Citation: Chen, Y.; Lu, Y.; Pan, Z.; Chen, J.; Shi, F.; Li, Y.; Jiang, Y. APIMiner: Identifying Web Application APIs Based on Web Page States Similarity Analysis. *Electronics* **2024**, *13*, 1112. https://orcid.org/10.3390/electronics13061112

Academic Editors: Tania Cerquitelli, Arkaitz Zubiaga, Giovanni Malnati and Genoveva Vargas-Solar

Received: 26 January 2024
Revised: 7 March 2024
Accepted: 13 March 2024
Published: 18 March 2024

1. Introduction

With the rapid development of the Internet, web applications have been integrated into all aspects of people's lives. As web applications become more widely used and their functions become more powerful, their inherent data and functional value also make them a target for attackers. Modern web applications are rich in interactivity and provide users with various APIs (Application Programming Interfaces) for data interaction.

However, the coin has two sides. The availability of various APIs for data interaction has provided users with convenience. However, it has also exposed web applications to more significant security threats due to the increased number of interfaces. For example, APIs can trigger XSS (cross-site scripting), SQL, remote code execution, and other client and server vulnerabilities. From the perspective of security analysts, the more APIs used for data interaction in a web application, the more likely it is that loopholes will be detected in the web application. Therefore, it is crucial to identify APIs in web applications as comprehensively as possible.

Existing studies on web application API identification are mainly based on the user manuals of web applications [1–3] and identify APIs by dynamically traversing web pages (e.g., using crawlers) [4–8]. These studies use methods such as regular matching to extract

APIs from user manuals or machine learning to generate a large number of test cases for guessing possible APIs. These methods struggle to identify APIs not listed in user manuals. Additionally, many web applications do not provide user manuals at all. Compared with extracting and generating web application APIs from user manuals, using a crawler to traverse web pages dynamically and extract APIs from the web pages can avoid the limitations of user manuals. However, developers use languages such as JavaScript, HTML, and CSS to implement highly complex user interfaces. The high complexity of the web front-end user interface and the similarity between these user interfaces pose a challenge for using crawlers to identify APIs in web applications.

Current methods for web page similarity analysis mainly include URL-based approaches [8], web page visual analysis [9,10], web page content hashing, and DOM (Document Object Model) structure comparison [11,12]. Although these similarity analysis methods have been proven to mitigate the negative impact of similar pages to some extent, they still have several shortcomings that make them not universally applicable. Since a large number of web applications have adopted the RESTful [13] specification or the MVC architecture [14], the URLs of different functions of such web applications are highly similar, which introduces a massive challenge to URL similarity analysis. Web page visual similarity analysis has limited usage scenarios, and only coarse-grained similarity analysis can be performed. When it is applied to API recognition scenarios, there will be situations where visual similarity occurs but the APIs contained are different. The web page content similarity analysis method has better applicability compared to other methods. However, the crawler needs to store and calculate the hash value of the web page content or structure. Oversized web page content can seriously affect the efficiency of the crawler. On the other hand, these methods usually use edit distance algorithms [15] to calculate the similarity. These algorithms do not consider the order relationship of the elements on the page. Moreover, the similarity analysis method based on the DOM tree structure is coarse-grained. In a web application, there may be situations where the DOM tree structure is similar but the specific content within the DOM tree nodes differs. For example, the API information contained in the nodes is different, which can cause the loss of some APIs. In addition, using web page views for similarity analysis is not appropriate for web API identification. Thus, it is urgent to design a more suitable web page similarity analysis method to reduce the impact of similar pages on the efficiency and accuracy of API identification.

To address the above challenges and issues, in this paper, we propose APIMiner, a framework for identifying the APIs of web applications by dynamically traversing web pages based on web page state similarity analysis. Specifically, when APIMiner accesses a web application, it first builds a web page model based on the HTML elements of the current web page. The web page model consists of triples $< URL\_link, Forms, JavaScript\_events >$. These three elements are the main causes of changes in page content. APIMiner uses this web page model to represent the state of the page. Then, APIMiner calculates the similarity of the data of the same element in different web page models, which can overcome the misjudgments caused by the different positions of data in the DOM tree. After calculating the similarity of each element, APIMiner judges the similarity of the page states based on the values of the similarity results and the weights of each element. Finally, from the different states of the page, APIMiner extracts the data interaction APIs on the page. In this approach, APIMiner traverses as many different web page states as possible, improving the coverage of web application APIs.

We conduct extensive experiments to verify the effectiveness of APIMiner by comparing it with state-of-the-art methods such as NDD [16] and mNDD [17]. The experimental results demonstrate that our method surpasses existing state-of-the-art methods in accurately and finely distinguishing between similar and dissimilar pages. We compare APIMiner with state-of-the-art tools (e.g., Enemy of the State [6], Crawlergo [18], and Wapiti3 [19]) for API identification. Regarding web application API identification, the experimental results show that APIMiner identifies an average of 1136 APIs in 10 web applications. In contrast, Enemy of the State identifies an average of 51 APIs in the same web applications, whereas Crawlergo identifies an average of 171 and Wapiti3 identifies an average of 296. APIMiner is 22.2, 6.64, and 3.84 times more effective than Enemy of the State, Crawlergo, and Wapiti3. APIMiner performs well in terms of the total lines of code executed in the ten tested web applications. On average, APIMiner executes 28,470 lines of code in each application, which are good results compared to the state-of-the-art tools. Enemy of the State averages 7578 lines, whereas Crawlergo and Wapiti3 execute an average of 18,060 and 15,293 lines, respectively.

The main contributions of this paper are summarized as follows:

- We propose APIMiner, a framework for identifying APIs in web applications by dynamically traversing web pages based on web page state similarity analysis.
- We design a new state representation of the web page and page state similarity analysis method. The new state representation of the web page similarity analysis method can effectively improve the identification of APIs, accurately and effectively reducing the impact of similar pages during traversal.
- We conduct extensive experiments to verify the effectiveness of APIMiner. The experimental results show that APIMiner achieves good performance in terms of the number of identified APIs (average of 1136) and code coverage (average of 28,470).

2. Background

2.1. Web APIs

APIs for data interaction are crucial for web applications, offering a uniform method for accessing data and features. These APIs facilitate communication between different applications or systems, supporting data sharing and integration. Various API styles are utilized, including RESTful, SOAP, JSON-RPC, and traditional web APIs. For instance, consider a web application that manages book information to illustrate the use of these web APIs.

RESTful APIs. RESTful APIs are a widely used web API type that uses the HTTP protocol to manage web resources. They operate on a resource-based concept, with resources represented by URLs and interacted with through HTTP methods like GET, POST, PUT, and DELETE. These APIs are stateless, treating each request independently, and are known for being lightweight, scalable, and straightforward. For a book information web application, such an API would include specific endpoints, as shown in Figure 1.

```
    GET /books: Retrieve a list of all books
    GET /books/{id}: Retrieve information about a specific book with the given ID
    POST /books: Create a new book with the provided information
    PUT /books/{id}: Update the information for the book with the given ID
    DELETE /books/{id}: Delete the book with the given ID
```

Figure 1. An HTTP request of a RESTful API.

SOAP APIs. SOAP APIs are web APIs that use XML for message encoding and follow standardized communication protocols, including XML message formats and HTTP or SMTP for transport. This paper focuses on SOAP APIs over HTTP, often chosen for enterprise applications due to their emphasis on security and reliability. While SOAP APIs may be more complex and require additional libraries, they offer robust error handling and security. As shown in Figure 2, in a web application managing book data, a SOAP API HTTP request would include an XML document with a SOAP envelope and a message body requesting details for a book with ID 12345.

JSON-RPC APIs. JSON-RPC APIs are lightweight web APIs that use JSON for data encoding. They enable remote procedure calls over HTTP or HTTPS, offering a straightforward implementation method. Ideal for mobile applications that need quick and efficient server communication, JSON-RPC APIs are widely utilized. As shown in Figure 3, a JSON-RPC API request to query a book would contain a JSON-RPC request object specifying the method ("getBook"), parameters (the book's ID), and a unique request ID.

```
POST /webservice HTTP/1.1
Host: example.com
Content-Type: text/xml; charset=utf-8
Content-Length: nnn

<?xml version="1.0"?>
<soap:Envelope xmlns:soap="http://schemas.xml
soap.org/soap/envelope/">
  <soap:Body>
    <GetBookRequest xmlns="http://example.com/
    bookservice">
      <BookID>12345</BookID>
    </GetBookRequest>
  </soap:Body>
</soap:Envelope>
```

Figure 2. An HTTP request of a SOAP API.

```
POST /api HTTP/1.1
Host: example.com
Content-Type: application/json
Content-Length: nnn

{
    "jsonrpc": "2.0",
    "method": "getBook",
    "params": 12345,
    "id": 1
}
```

Figure 3. An HTTP request of a JSON-RPC API.

Traditional web APIs. Traditional web APIs can be categorized into form-based APIs and GET request APIs with parameters. Form-based APIs utilize HTML forms for data submission to a server, processing user-submitted forms and responding accordingly. As shown in Figure 4, an HTTP request for a book query via a form-based API involves the browser sending this request following form submission. On the other hand, traditional GET request APIs with parameters transmit data through URL parameters, where the data are encoded as key-value pairs in the URL, separated by ampersands. An example of a book query using a traditional GET request API would be "http://example.com/getBook.php?bookid=12345".

```
POST /getBook.php HTTP/1.1
Host: example.com
Content-Type: application/x-www-form-urlencoded
Content-Length: nnn

bookid=12345&submit=search
```

Figure 4. An HTTP request of a form-based API.

2.2. Web Page State

The state of a web page refers to its content and structure at any given moment, including text, images, form values, user interactions (such as clicks and scrolls), and elements dynamically modified by client-side scripts like JavaScript. This state is crucial for delivering rich user experiences and interactive web applications, as it allows the page to adapt its content and behavior in response to user interactions. Consider a simple web application with only three pages: index, about, and edit. Figure 5 presents a case of state change for this web application. Taking the S_0 state of the `index` page as the root state, it includes the URL link to access the `about` page. When the user fills in the login information and submits the login form, the `index` page transitions from the S_0 state to the S_1 state. In the current state of the `index` page, in addition to the original link to access the `about` page, a new URL link to access the `edit` page appears. Importantly, the link to the `edit` page appears only in state S_1.

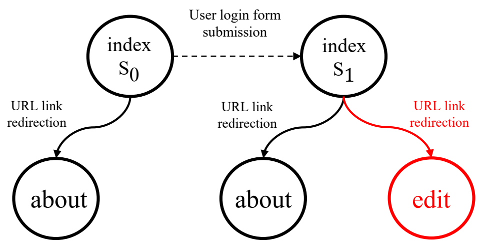


Figure 5. State changing of web pages.

3. Overview

Figure 6 presents an overview of APIMiner, which is composed of three main parts: state representation, page state similarity analysis, and API identification. APIMiner aims to overcome the challenge of page similarity analysis and cover as many web application page states as possible to identify and extract APIs.

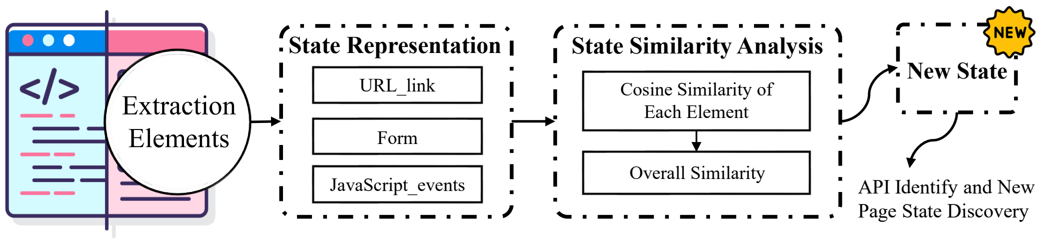

Extraction Elements
State Representation
URL_link
Form
JavaScript_events
State Similarity Analysis
Cosine Similarity of Each Element
Overall Similarity
NEW
New State
API Identify and New Page State Discovery

Figure 6. Overview of APIMiner.

3.1. State Representation

Unlike the input interface of a binary program, the data interaction interface of a web application exists mainly on the web application page [20]. The user typically interacts with a server-side web application by clicking controls on the page, filling in a form, etc. Since web applications cannot ensure that each interactive interface has sufficient security detection capabilities for data input by the user, improving the coverage of these interfaces can increase the probability of triggering vulnerabilities.

Traversing the target web applications as comprehensively as possible is the core challenge for API identification. 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 event is triggered). Therefore, it is necessary to capture the state changes of every web page during traversal. Otherwise, some web pages will not be visited, and even worse, some APIs will be missed. To better emphasize the differences between web page states, we propose a more precise method for representing web page states.

Whether the content of the web page changes is mainly affected by three types of elements. The first element is the URL link on the web page. Figure 7 shows part of the front-end code of the navigation bar of the WordPress administrator interface. The data in the `href` field in the two $< a >$ tags are the relative URLs that access the Home and Update user interfaces, respectively. When the user clicks Home, the WordPress user interface redirects to "https://domain:port/index.php", and the page's content changes. The URL addresses in these tags are mainly used to switch web pages or functions. They have the same domain name as the target website and do not point to resource-type files, such as JavaScript, CSS, images, videos, etc. In the front-end pages of web applications, there are many similar tags that include such URLs, such as $< a >$, $< link >$, $< iframe >$, etc., and they use `href` or `src` fields for page redirection. It is worth noting that there are also a large number of URLs for accessing specified functions in the front-end $< script >$ tags of web applications, such as the data in `location.href`. Once these URLs for page or function switching are accessed, the content of the web page == changes.

```
<ul class='wp-submenu wp-submenu-wrap'>
  <li class='wp-submenu-head' aria-hidden=
  'true'>Dashboard</li>
  <li class="wp-first-item current">
    <a href='index.php' class="wp-first-item
    current" aria-current="page">Home</a>
  </li>
  <li>
    <a href='update-core.php'>Updates</a>
  </li>
</ul>
```

Figure 7. Part of the front-end source code of the WordPress administrator interface.

The second element is the form submission on the web page. Forms are one of the main ways for users to interact with web applications. Take the WordPress administrator login interface as an example. When an administrator enters the correct login credentials into the form and submits the data, the web application authenticates the user and redirects them to the administrator interface. In this way, forms provide a crucial functionality in web applications, allowing users to input and submit data to the server for processing and response.

The third element is the execution of JavaScript events on the web page. With the advent of highly complex user interfaces, web developers often implement intricate designs using languages such as JavaScript, HTML, and CSS. These interfaces employ various JavaScript events, such as mouse clicks, keyboard inputs, and touch events, to provide users with a rich and interactive experience. Using these events, web pages can dynamically render content, update functions and data in real time, and provide various interactive elements, such as sliders, pop-ups, and menus. The triggering of these JavaScript events often changes the page content, which may include updating functions that introduce new URL addresses or APIs.

In summary, APIMiner utilizes these three types of elements to build a comprehensive web page model based on the URL, forms, and JavaScript events of the current web page. That is, the web page model is represented as $< URL\_link, Form, JavaScript\_events >$. By doing so, it can accurately analyze the behavior and state of web applications and improve the efficiency and accuracy of API identification.

URLs in web pages mainly include absolute URLs and relative URLs. An absolute URL (Uniform Resource Locator) specifies the exact location of an internet resource, encompassing the protocol, domain name, and path. This complete web address details the server and directory location of the resource, ensuring precise identification and access. On the other hand, a relative URL provides a partial web address that points to a resource relative to the current web page's URL. Relative URLs are commonly used when linking to resources within the same website, and they are shorter and easier to manage. APIMiner matches and extracts all URLs from the $< a >$, $< link >$, $< iframe >$, and $< script >$ tags. APIMiner performs data cleaning on absolute URLs, excluding non-local URLs and resource file URLs. When APIMiner encounters forms on web pages, it extracts the key data in the form and uses a unified format to describe the form, as shown in Figure 8. Specifically, APIMiner extracts the data interaction method (mainly including GET and POST), the URL of the action field, fields, and corresponding values in post params, such as data from the `id` and `value` fields in the $< input >$ tag. To comprehensively obtain all the event information on the current page, APIMiner uses the `getEventListeners()` command in Google Chrome. This command enables APIMiner to obtain all event information on the current page, including information on events triggered by JavaScript code embedded in the HTML code, such as "onclick" and "onload" events in the $< button >$ and $< body >$ tags. By obtaining event information, APIMiner can accurately identify and describe the events on the page, providing valuable information for subsequent testing and analysis. The triplet representation after extracting the three types of data is shown in Figure 9.

```
{   'method': 'GET/POST',
    'action': 'Relative URL',
    'postdata': {
        'key_1': 'value_1',
        'key_2': 'value_2',},}
```

Figure 8. The format of a form's key data.

```
"web page URL": {
        "Javascript events": [
            "Javascript events_1",
            "Javascript events_1",
            "Javascript events_1",
            ...],
        "form": [
            "form_1",
            "form_2",
            ...],
        "url": [
            "URL_1",
            "URL_2",
            "URL_3",
            ...]}
```

Figure 9. Concrete representation of triplets.

3.2. State Similarity Analysis

In the process of web traversing, APIMiner is responsible for modeling each visited page state. When there is a URL link redirection, form submission, or JavaScript event trigger behavior on the page, the page is remodeled to reflect these changes. In order to determine whether the updated page should be further processed or crawled, APIMiner conducts a similarity analysis between the new and previous page states. By analyzing the similarity between the old and new page states, APIMiner can identify whether the changes in the page warrant further data processing and traversing. This helps improve the accuracy of data extraction and the efficiency of traversing. Additionally, conducting similarity analysis can help prevent APIMiner from entering an infinite loop, where duplicate web pages are repeatedly visited.

Existing similarity analysis methods used in web pages have certain limitations. For example, consider a page in the book management system "/getbook.php?bookid=1" and its two sub-URLs "/getbook.php?bookid=1&actionid=1" and "/getbook.php?bookid=1 &actionid=2". The "actionid=1" and "actionid=2" correspond to the editing information and viewing functions of the book, respectively. When using the URL similarity analysis method, the two URLs are usually clustered together and judged as similar. Using edit distance for similarity analysis may result in incorrect judgments due to the sequence relationship between elements and their positions in the DOM tree. For instance, the sequence of three elements (A, B, and C) could be either "ABC" or "CBA", leading these methods to classify such pages as dissimilar. Moreover, the similarity analysis method based on the DOM tree structure is coarse-grained. In web applications, situations may arise where the DOM tree structure is similar but the specific content within the DOM tree nodes differs. For example, the API information contained in the nodes is different, which can cause the loss of some APIs. In addition, using web page views for similarity analysis is not appropriate for web API identification. Therefore, in order to overcome the applicability and misjudgment of existing similarity analysis methods, we have designed a new similarity analysis method. The similarity analysis process is shown in Algorithm 1.

Algorithm 1 Traverse and similarity analysis method

```
1: function START($start_url$)
2:     $start\_state \leftarrow PageStateModeling(start_url)$
3:     $Traverse(start\_state)$
4: end function
5: function TRAVERSE($start\_state$)
6:     $queue = deque([start\_state])$
7:     $visited\_state = set()$
8:     while $queue$ do
9:         $current\_state = queue.popleft()$
10:        if $current\_state \notin visited\_state$ then
11:            $new\_state\_list \leftarrow StatesAnalysis(current\_state)$
12:            $visited\_state.add(current\_state)$
13:            for $new\_state \in new\_state\_list$ do
14:                if $SimilarityAnalysis(visited\_state, new\_state, weights, threshold)$ then
15:                    $queue.append(new\_state)$
16:                end if
17:            end for
18:        end if
19:    end while
20: end function
21: function SIMILARITYANALYSIS($state\_list, new\_state, weights, threshold$)
22:     for $current\_state \in state\_list$ do
23:         $URL\_similarity \leftarrow Similarity_{element}(current\_state.url, new\_state.url)$
24:         $Form\_similarity \leftarrow Similarity_{element}(current\_state.form, new\_state.form)$
25:         $JSEvents\_similarity \leftarrow Similarity_{element}(current\_state.jsevent, new\_state.jsevent)$
26:         $OverAll\_similarity \leftarrow weights.url * URL\_similarity + weights.form * Form\_similarity + weights.jsevent * JSEvents\_similarity$
27:         if $OverAll\_similarity > threshold$ then
28:             return False
29:         end if
30:     end for
31:     return True
32: end function
```

Specifically, first, APIMiner vectorizes the three types of elements in the current page state through the word frequency vectorization method. Take the example of URL links. For each unique URL link, APIMiner counts the number of times it appears on the page and then uses this count as the vector value. For instance, if the array of URL link elements is [$URL_1, URL_2, URL_3, URL_1, URL_1$], where URL_1 appears three times on the page, the vectorized URL link element is [3,1,1]. Second, APIMiner calculates the cosine similarity values between the three types of elements in the current page state and the corresponding category elements in the previous page state. The cosine similarity calculation formula is shown in Formula (1). If the calculation result is close to 1, it means that the two sets of data are more similar; if the result is closer to 0, it means that the two sets of data are less similar. In this way, the influence of the position of the element in the web page structure is ignored. For example, the URL link elements in these two page states are [$URL_1, URL_2, URL_3, URL_1, URL_1$] and [$URL_1, URL_1, URL_4, URL_4, URL_5$]. Calculated using Formula (1), the cosine similarity of the URL link elements in these two page states is 0.6030, meaning there are certain differences in the URL link elements in these two page states.

$$\begin{aligned} Similarity_{element} = \cos\theta &= \frac{X \cdot Y}{|X| \times |Y|} \\ &= \frac{\sum\limits_{i=1}^{n} (X_i \times Y_i)}{\sqrt{\sum\limits_{i=1}^{n} (X_i)^2} \times \sqrt{\sum\limits_{i=1}^{n} (Y_i)^2}} \end{aligned} \tag{1}$$

Finally, APIMiner computes the overall similarity between the two page states based on the preset weights of the three elements using Formula (2). In web pages, the degree of change in the page content resulting from URL link redirection, form submission, or JavaScript event triggering varies, where URL Link > Form > JavaScript events. Concretely, URL link redirection can have a profound impact on the web page's state. When a user clicks on a URL link, it may trigger a redirection that leads to the entire web page reloading. This action can cause significant changes to the page content, as the reload may fetch entirely new content from the server. Form submission is another action that can result in substantial changes to a web page's state. When a form is submitted, it typically triggers a server-side response, which then alters the state of the web page. This change could manifest as a partial rendering of the page content, where only certain elements of the page are refreshed. On the other hand, it could also lead to a complete page reload, similar to what happens during URL link redirection. JavaScript events, in contrast, typically result in more nuanced changes to a web page. These events often lead to dynamic updates of specific sections of the page content without instigating a total page reload. For instance, a JavaScript event might result in a change to a drop-down menu, the appearance of a pop-up window, or the dynamic loading of additional content on the page. However, it is also worth noting that not all JavaScript events lead to noticeable changes in the page content. Some might trigger actions that are not visible to users, such as logging data or sending background requests to the server. We assign different weights to these elements based on their respective impacts on changes to web page content. The overall similarity calculation formula for different page states is shown in Formula (2), where $\omega_{url} + \omega_{form} + \omega_{js} = 1$.

$$\begin{aligned} Similarity(A, B) = &\ \omega_{url} \times Similarity_{url} \\ &+ \omega_{form} \times Similarity_{form} \\ &+ \omega_{js} \times Similarity_{js} \end{aligned} \tag{2}$$

APIMiner sets a similarity threshold T. If the computed overall similarity between the A and B page states $Similarity(A, B) > T$, then A and B are considered similar. Conversely, if $Similarity(A, B) < T$, then A and B are considered dissimilar. By analogy, if the current page state is different from the previous page state, it is defined as a new page state. APIMiner identifies the API and explores the next new page state based on this new page state. Employing this similarity analysis method, APIMiner traverses the web application user interface as comprehensively as possible using breadth-first traversal [21]. Taking Figure 10 as an example, the S_0 state of the `index` page is the root state. After submitting a form and triggering an "onload" event, APIMiner discovers two new page states, S_1 and S_2, of the `index` page. By traversing the S_1 and S_2 states of the `index` page, APIMiner can discover the S_0 state of the `settings` page and the new S_3 state of the `index` page from the S_1 state of the `index` page. Similarly, it can detect the S_0 state of the `view` page and the S_4 state of the `index` page from the S_2 state of the `index` page. APIMiner continues to traverse the web application by exploring the newly discovered page states until no additional page states are found, completing the traversal of the entire application.

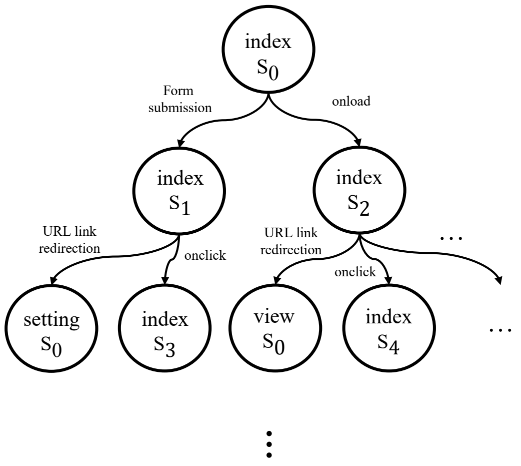


Figure 10. Traversing the various page states of the web application.

3.3. API Identification

As APIMiner traverses the page states, it identifies and extracts APIs for each unique page state. APIMiner is able to extract APIs that conform to the RESTful API, SOAP API, and JSON-RPC API specifications in the URL element of the page state. Additionally, APIMiner extracts traditional HTTP request APIs from URLs that contain HTTP GET request parameters, such as "http://domain:port/administrator/index.php?option=com_categories&extension=com_users" in Joomla, which are also located in the URL element of the page state. Furthermore, APIMiner constructs and identifies traditional HTTP request APIs generated by form submission based on the data structure of the form element in the page state. Specifically, APIMiner constructs the HTTP request API templates according to the HTTP request headers specification [22]. Each piece of API data includes the URL address, HTTP request method, HTTP request header (e.g., Host, Cookie), and HTTP POST body [23], as shown in Figure 11.

```
{
    "URL": "",
    "Method": "",
    "Headers": {
        "Host": "",
        "Cookie": "",
        "User-Agent": "",
        "Referer": "",
    },
    "POST Body": "",
}
```

Figure 11. The format of a piece of data in a web application API.

After completing the web application page state traversal and API identification, APIMiner must clean the extracted API data by removing duplicate API entries. This process ensures that the API data are accurate and concise, without unnecessary duplication, so that they can be used effectively for subsequent analysis and testing.

4. Evaluation

In this section, we evaluate the performance of APIMiner in identifying web application APIs and compare it to state-of-the-art similarity analysis methods and tools. We aim to answer the following research questions in the evaluation of APIMiner:

- RQ1. How capable is APIMiner in page similarity analysis? How does it compare with existing similarity analysis methods?

- RQ2. How effective is APIMiner's augmentation technology in identifying web application APIs? How does the number of web application APIs identified by APIMiner compare to state-of-the-art tools, such as Crawlergo and Wapiti3?
- RQ3. How does the coverage of APIMiner for web applications compare with existing methods?

4.1. Experimental Setup

Benchmark Applications. Table 1 shows information about the 10 web applications used as the benchmarks, as well as their specific versions. These 10 web applications were selected to cover a wide range of functionalities and APIs commonly used by web users. The benchmark set includes various types of web applications, such as e-commerce platforms, social networking sites, and forums. Additionally, the web applications were chosen from different sources, including the NAVEX test set [24], popular CMS (content management systems) applications listed by W3Techs [25], and high-scoring CMS projects on GitHub [26]. By including web applications from various sources, the benchmark set provides a comprehensive evaluation of APIMiner's capabilities in identifying APIs and vulnerabilities across a diverse range of web applications.

Table 1. The web applications used for testing.

#	Application	Version	LoC	#	Application	Version	LoC
0	Elgg	2.3.10	96,582	5	Joomla	3.9.3	33,949
1	Subrion	4.2.1	319,112	6	XE	1.11.2	145,169
2	CMSMadeSimple	2.2.9.1	310,316	7	WordPress	5.0.3	1,249,842
3	MyBB	1.8.19	237,667	8	phpBB	3.2.5	193,720
4	Backdrop	1.12.1	73,739	9	Drupal	8.6.9	20,512

Environment. To evaluate our system, we used state-of-the-art scanners that have been widely adopted for exporting identified web application APIs, allowing us to perform a direct comparison. Specifically, we evaluated APIMiner against Enemy of the State, Crawlergo, and Wapiti3. Enemy of the State is a popular state-of-the-art academic scanner that utilizes a state inference approach during web page traversal to build a comprehensive web application state machine model. It detects server state changes by analyzing responses to identical requests. Crawlergo is a browser crawler that hooks key positions of the whole web page during the DOM rendering stage, automatically fills and submits forms, triggers intelligent JS events, and collects as many entries exposed by the website as possible. We choose the latest version, Crawlergo 0.4.4, for experiments. Wapiti3 is a powerful, free, and open-source web application vulnerability scanner that supports the export of identified web API data. After years of update iterations, it is deeply loved by vulnerability researchers. We installed Wapiti3 version 3.1.7 for experimental comparison. We conducted experiments on an Ubuntu 18.04 with an Intel Core i7-9750 (2.60 GHz) CPU and 32 GB of RAM. The target web applications were deployed using the official Docker image of Ubuntu 18.04, with Apache 2.4.29, PHP 7.2.24, and Xdebug 2.6.0 installed. It is important to stress that the environments for APIMiner and these tools were the same to ensure fairness in the evaluation.

4.2. Similarity Analysis Capabilities

The experiment described below was conducted to verify the validity of our similarity analysis method and identify the ideal weight distribution and threshold for our web page similarity analysis method. Inspired by the study of Kostas et al. [17], we built the test data set using posts in WordPress. In total, we prepared the following three data sets, two of which consisted of pages we manually compiled:

(1) The first set included pages with similar URLs, functionalities, and APIs intended to be clustered. The set consisted of post pages with highly similar URLs, forms, and JavaScript events but different article content in WordPress. It is worth mentioning that the order of the three types of elements may be different. An example of a post is shown in Figure 12.

(2) The second set consisted of pages with similar access URLs but different APIs, which should not be clustered. These post pages contained completely different or partially similar URLs, forms, and JavaScript events. An example of a post is shown in Figure 13.

(3) The third set comprised pages with completely distinct URLs and functionalities, which should not be clustered together. This set consisted of WordPress pages where users were not logged in.

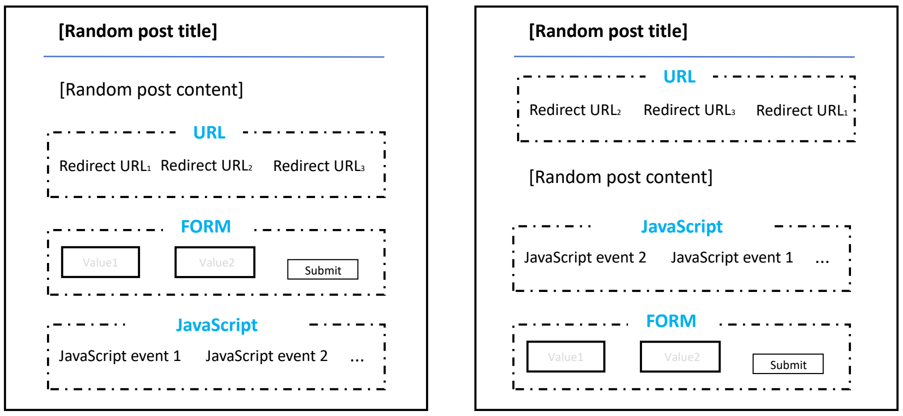

[Random post title]
[Random post content]
URL
Redirect URL$_1$ Redirect URL$_2$ Redirect URL$_3$
FORM
Value1
Value2
Submit
JavaScript
JavaScript event 1 JavaScript event 2 ...
[Random post title]
URL
Redirect URL$_2$ Redirect URL$_3$ Redirect URL$_1$
[Random post content]
JavaScript
JavaScript event 2 JavaScript event 1 ...
FORM
Value1
Value2
Submit

Figure 12. SET 1 post template.

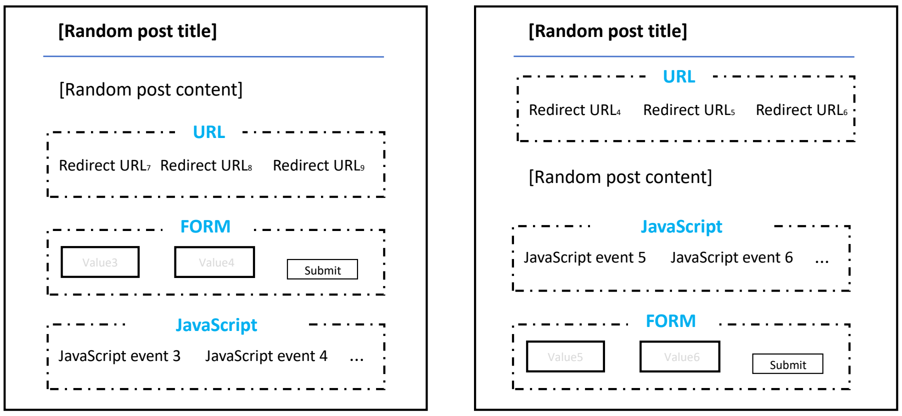
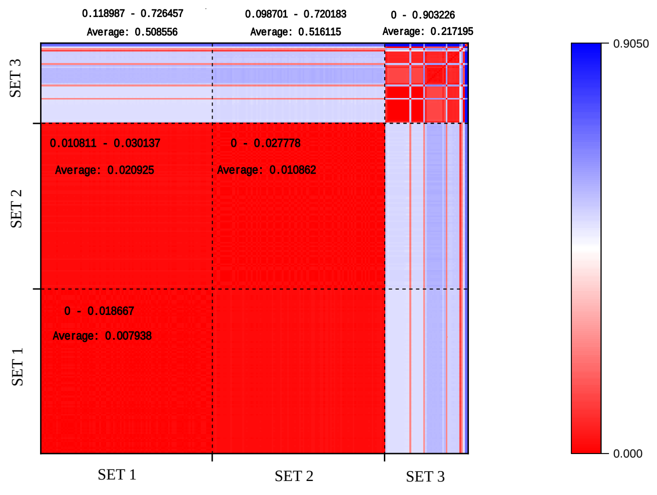

[Random post title]
[Random post content]
URL
Redirect URL$_7$ Redirect URL$_8$ Redirect URL$_9$
FORM
Value3
Value4
Submit
JavaScript
JavaScript event 3 JavaScript event 4 ...
[Random post title]
URL
Redirect URL$_4$ Redirect URL$_5$ Redirect URL$_6$
[Random post content]
JavaScript
JavaScript event 5 JavaScript event 6 ...
FORM
Value5
Value6
Submit

Figure 13. SET 2 post template.

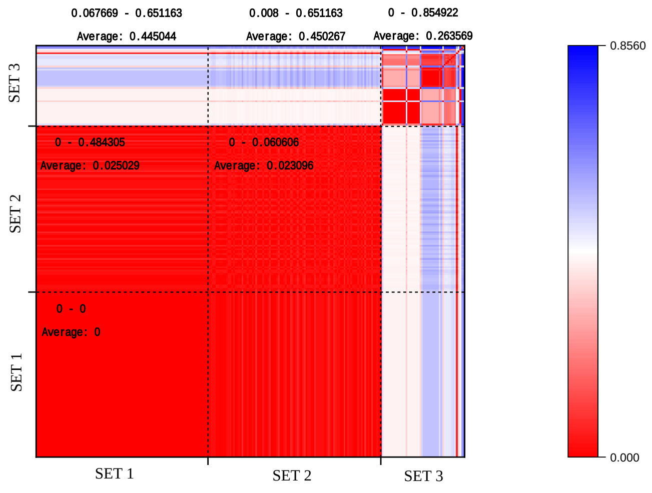
Additionally, the three data sets were not similar to each other. For addressing RQ1, given the similarity of the web page URLs in both SET 1 and SET 2, utilizing a URL-based similarity analysis method was not appropriate. Consequently, this paper abstained from conducting an experimental comparison of this method type. We compared the similarity analysis method proposed in this paper with state-of-the-art similarity analysis methods, including NDD (Normalized DOM-edit Distance) and mNDD [17].

The NDD method, proposed by Vissers et al. [16], operates by parsing an HTML string into a tree representation of the DOM nodes. It then establishes the difference between two trees by computing the edit distance. The mNDD method, a variant of NDD, was proposed by Kostas et al. [17]. In the process of constructing corresponding trees from the DOMs of pages, mNDD recursively discards leaf nodes devoid of functionality (e.g., tags for line breaks, paragraphs, spans, divs, or font formatting) while preserving nodes associated with functionality (e.g., scripts, forms, iframes, buttons, and inputs). Both methods employ the edit distance algorithm. In this context, a smaller edit distance between two page states implies a higher degree of similarity between them. We obtained the NDD and mNDD algorithms from the open-source projects of Vissers et al. [16] and Kostas et al. [17] for the experimental evaluation.

The heatmap in Figure 14 shows the experimental results obtained with NDD, mNDD, and APIMiner in the similarity analysis of the data set. Specifically, Figure 14a shows the results of NDD's similarity analysis of the three sets. We analyzed the experimental results based on the threshold of 0.18 set in the work by Vissers et al. [16] and found that NDD accurately judged all web pages in SET 1 as similar, with an average similarity value of 0.007938 for the pages in SET 1. When performing similarity analysis on SET 2, NDD judged that all pages in SET 1 and SET 2 were similar, with a maximum similarity value of 0.030137 between the pages in SET 1 and SET 2. Moreover, NDD could not distinguish between the pages in SET 2, with similarity values ranging from 0 to 0.027778, all smaller than the threshold. For SET 3, since these pages were all taken from WordPress, the DOM structures of some of them were similar, resulting in some pages in SET 3 being judged as similar by NDD, even though there were certain differences in their page content and functions.

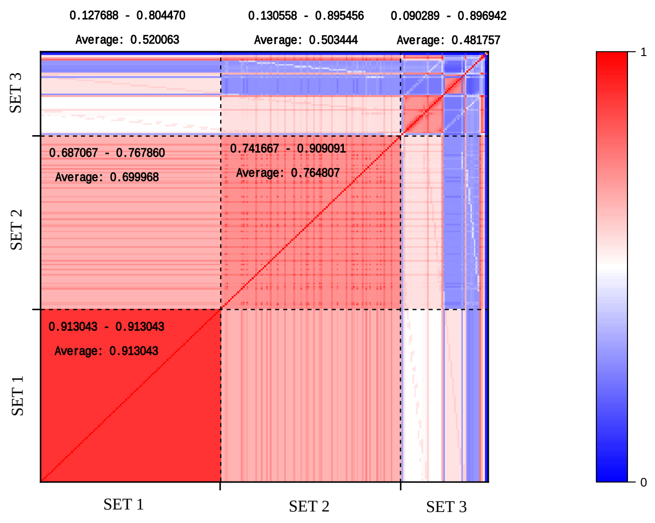


(**a**)



(**b**)

Figure 14. *Cont.*



(**c**)

Figure 14. Comparison of the similarity analysis capabilities of the three methods. The value data range in the figure represents the highest and lowest similarity values between two data sets, whereas the average represents the average similarity value of all pages between two data sets. (**a**) NDD: The closer the similarity value to 0, the redder the color, indicating that the two pages are more similar. (**b**) mNDD: The closer the similarity value to 0, the redder the color, indicating that the two pages are more similar. (**c**) APIMiner: The closer the similarity value to 1, the redder the color, indicating that the two pages are more similar.

Figure 14b shows that mNDD outperformed NDD in the similarity analysis of SET 2. We analyzed the experimental results using the threshold value of 0.09 established in the study by Kostas et al. [17]. Similar to NDD, mNDD accurately judged all pages in SET 1 as similar, with a similarity value of 0 for all pages in SET 1. For SET 2, mNDD judged 49% of pages in SET 2 as similar to all pages in SET 1, with similarity values ranging from 0 to 0.484305. Similar to NDD, mNDD could not distinguish between the pages in SET 2, with similarity values ranging from 0 to 0.060606, all less than the threshold of 0.09. mNDD performed better in SET 3 compared to NDD.

Figure 14c shows the similarity analysis results obtained with APIMiner. It can be seen that APIMiner accurately distinguished between the three data sets. In the similarity analysis method of APIMiner, we assigned weights of 0.5, 0.4, and 0.1 to the URLs, forms, and JavaScript events based on their impact on page changes. According to APIMiner's calculations, the minimum similarity value among the pages in SET 1 was 0.913043. The maximum similarity value between the pages in SET 1 and SET 2 was 0.767860449, whereas the highest similarity value among the pages in SET 2 was 0.909091. The maximum similarity value among the pages in SET 3 was 0.906745. Based on the calculation results of APIMiner, we set the threshold T of the similarity analysis method to 0.91, which enabled accurate differentiation between the three sets. The experimental results of the similarity analysis fully illustrate the effectiveness of our proposed similarity analysis method, which outperformed state-of-the-art page similarity analysis methods.

4.3. Web Application API Identification

Table 2 details the number of APIs identified and the code coverage achieved by Enemy of the State, Wapiti3, Crawlergo, and APIMiner in 10 popular modern web applications.

Number of APIs Identified. The *No.* columns indicate the number of unique web application APIs discovered by the four tools. The experimental results in this column address RQ2. It can be seen in Table 2 that the number of APIs identified by APIMiner in the 10 applications was significantly greater than those identified by Enemy of the State, Wapiti3, and Crawlergo. According to the experimental results, Enemy of the State failed to extract the APIs during tests against Subrion, Backdrop, XE, and phpBB. After a detailed analysis, we found the cause to be related to the outdated version of the dependent library used by Enemy of the State. This led to its inability to complete page traversal and API identification in the web application that used "jquery.min.js". According to the experimental results for MyBB, both Enemy of the State and Wapiti3 reported a significantly higher number of identified APIs compared to APIMiner. The reason for this disparity is that Enemy of the State and Wapiti3 do not include a page similarity analysis method, which led to the extraction of a large amount of similar data. Specifically, there is a calendar view function in MyBB, which led Enemy of the State and Wapiti3 to fail to promptly judge the similarity of the calendar page after entering this function, resulting in the extraction of large quantities of calendar APIs. We manually deduplicated the API data for MyBB provided by Enemy of the State and Wapiti3, as shown outside the brackets on the MyBB line in Table 2. The effective data extracted by Enemy of the State and Wapiti3 were lower than that of APIMiner. This case clearly illustrates the importance and effectiveness of page similarity analysis.

Table 2. Performance comparison between APIMiner and state-of-the-art methods in identifying web APIs.

#	**Web Application**	**Enemy**		**Wapiti3**		**Crawlergo**		**APIMiner**	
		No.	**ELoC**	**No.**	**ELoC**	**No.**	**ELoC**	**No.**	**ELoC**
0	Elgg	3	7257	132	14,840	149 (3462)	14,567	**175**	**16,760**
1	Subrion	0	1784	163	6000	269 (493)	5368	**511**	**12,515**
2	CMSMadeSimple	7	1446	36	10,915	53 (57)	15,732	**138**	**20,724**
3	MyBB	105 (1342)	9237	162 (463)	9983	147 (166)	12,250	**293**	**17,293**
4	Backdrop	0	1054	109	14,733	500 (803)	30,613	**4222**	**42,526**
5	Joomla	49	8702	200	17,199	68 (113)	14,059	**243**	**31,231**
6	XE	0	5629	99	12,656	121 (355)	16,052	**286**	**23,380**
7	Wordpress	140	17,203	184	31,401	79 (342)	22,766	**1304**	**41,322**
8	phpBB	0	3888	59	5245	59 (344)	11,342	**77**	**12,200**
9	Drupal	212	19,579	1824	29,953	274 (1380)	37,941	**4119**	**66,751**

The data in the column indicate the number of identified APIs (No.) and the total lines of code executed (ELoC). Values in bold indicate instances where the tool identified the most APIs or executed the most lines of code in the target application. Data within the brackets are those reported by the tool, whereas data outside the brackets are the results after data validity cleaning.

Regarding the experimental results of Crawlergo, it achieved good results in API identification in each web application, as indicated by the data within the brackets in the Crawlergo column in Table 2. However, The API data reported by Crawlergo contained a significant amount of invalid information. During the traversal process, Crawlergo incorrectly spliced the "contentType" data in JavaScript with the URL [27] as an API for extraction. We determined the validity of these APIs by replaying their requests and analyzing the HTTP response status code. For instance, if the HTTP response status code from replaying an API request was 403 or 404, it indicated that this API was invalid. Using this method, we performed data cleaning on the results reported by Crawlergo to ensure the validity of the APIs. The cleaned data are presented outside the brackets in the Crawlergo column.

For the experimental results of APIMiner, we used the same data cleaning method as Enemy of the State, Wapiti3, and Crawlergo. Table 2 shows that the number of APIs recognized by APIMiner significantly surpassed those of the other three state-of-the-art tools. Across these 10 web applications, the number of APIs recognized by APIMiner increased by at least 0.17 times, up to 57.33 times, with an average increase of 7.96 times, excluding the case where the Enemy of the State was unable to perform a scan.

Code coverage. The *ELoC* column indicates the total lines of code executed by the four tools during testing of the web applications, which is used to represent the code coverage of these tools. We used the open-source code coverage calculation environment [28] of the research work by Kostas et al. [17] to determine the code coverage in our experiments. The experimental data in the *ELoC* column address RQ3. It is clear that APIMiner achieved superior code coverage across all tested web applications. Excluding the web applications that Enemy of the State cannot normally crawl, compared to the three state-of-the-art tools, APIMiner's code coverage increased by at least 7.56%, up to a maximum of 1333.20%, with an average increase of 142.72%. This demonstrates that in many cases, APIMiner is capable of exploring more page states in web applications compared to existing state-of-the-art tools, thus increasing the likelihood of accessing key functions. This is also stronger proof that APIMiner can identify more APIs in web applications.

5. Discussion

Although APIMiner achieves good performance in page similarity analysis, identification of web application APIs, and code coverage, it still has limitations that need further research.

Identity switching. APIMiner, Crawlergo, and Wapiti3 all employ website cookie sharing for identity authentication on websites. The shared cookies from websites usually only contain the identity state of one role, such as the site administrator. However, modern web applications often support multiple roles, like community-type web applications encompassing multiple roles like site administrators and community users. In these cases, identity authentication through website cookie sharing is unable to facilitate the transition between multiple website roles, consequently missing APIs under other role states. In future work, we aim to explore identity switching among different roles on websites to maximize coverage of various page states within web applications.

Fine-grained distinction of JavaScript events. Considering that most JavaScript events within a web page do not generally cause state changes, APIMiner assigns a lower weight to JavaScript events during the analysis of page state similarity. Nevertheless, it is undeniable that some JavaScript events can result in substantial modifications in page content upon triggering. These events could, for example, initiate a major content change, transform the displayed information, or even modify the structure of the web page. These changes can substantially alter the user experience and, thus, the "state" of the page from an interaction perspective. As a result of this understanding, our future work will consider a more detailed and nuanced division of JavaScript events. This fine-grained division would allow us to better distinguish between events that lead to minor or no real changes and those that cause significant page transformations. By achieving this, we aim to make APIMiner more effective in understanding and interacting with dynamic web pages, thus improving its overall functionality and performance.

Single-page applications. Single-page applications represent a mainstream form of contemporary web applications and are a target for our future work. In the evaluation of this manuscript, we focus on CMS-type web applications. In future research, we will explore improving the universality of APIMiner to explore its effectiveness in single-page web applications.

Optimal Algorithm. In APIMiner, we use the cosine similarity algorithm to calculate the similarity of each type of element, although various other algorithms could be also utilized for vector similarity comparison. This paper concentrates on highlighting the distinct impacts of three types of elements on the state of web pages and does not delve into the optimal vector calculation algorithm. In future research, we plan to investigate which algorithms yield superior performance.

6. Related Works

This section discusses the related works. Given the widespread use of web applications, identifying and extracting APIs for subsequent vulnerability detection has become a popular and important research topic. Existing studies on API identification are mainly based on the user manuals of web applications [1–3,29–32] and identify APIs by dynamically traversing web pages [4–8]. In dynamic web page traversal for identifying web application APIs, existing studies utilize similarity analysis methods to mitigate the impact of similar pages on the API identification process [8–12].

Web application API identification. Both Restler [2] and Schemathesis [3] extracted web application APIs based on the development user manual or the API reference (such as OpenAi and GraphQL) of the web application [29]. They aimed to find the correlation between these APIs to uncover deeper vulnerabilities in subsequent fuzzing. H.A. Grent [1] and Gao et al. [30] statically identified and extracted web application APIs from API references using rule matching and then used machine learning techniques to analyze and infer the relationships between parameters in the API to generate more APIs. The effectiveness of the generated APIs was evaluated through probing. However, the APIs extracted or generated by these methods rely on user manuals or API references, and most web applications do not provide user manuals or API references. In comparison, our work focuses on identifying and extracting web application APIs directly from web pages without relying on user manuals or API references. As early as 2006, Kals et al. [33] designed a simple dynamic web application vulnerability scanner that can access the page and extract the HTML data, thereby identifying the API and injecting malicious data into the API for security detection. Crawljax [5] is a dynamic analysis state-aware crawler that explores AJAX-based client-side applications to identify APIs. It uses a DOM tree to represent the state of the page and compares the edit distance of the string representation of the DOM trees to compare states. Enemy of the State [6] utilizes a state inference approach during web page traversal to build a comprehensive web application state machine model. It detects server state changes by analyzing the responses to identical requests. We offer a more comprehensive and lightweight representation of the page state compared to these methods. By analyzing key elements that trigger changes in web pages and using these to represent the page state, APIMiner does not need to store the entire page content or the full DOM tree. Through finer-grained page state representation, we can more effectively detect changes in page states and cover more web application page states, leading to improved API coverage.

Web page similarity analysis. Similarity analysis is also widely used, which aims to mitigate the impact of similar pages on the API identification process, such as causing inefficiency and entering an infinite loop of repeat visits to duplicate pages. Some previous studies used URL similarity analysis [8,34,35] for deduplication and set the crawl depth to avoid loop traversal. Moreover, some studies used web page content for similarity analysis, such as calculating the hash or edit distance of the DOM tree or the page structure to judge whether web pages are similar [11,12,17,36,37]. In addition, some previous studies [9,10,38–41] used web page visuals for similar page analysis, such as the detection of similar phishing pages. Despite being successful in their respective goals, such approaches still have limitations. Specifically, there are a large number of similar URLs in a web application, and there are often URLs with similar paths and similar query parameters, which provide entirely different functions. Avoiding an infinite loop of repeat visits to duplicate pages by setting the traversal depth can also lead to missing a lot of APIs. Regarding methods that calculate the hash or edit distance of the DOM tree or the page structure, on the one hand, storing a large amount of page information can impact traversal efficiency, especially when the content of a web page is extensive. On the other hand, these methods do not consider the position and order of elements on the page. Visual similarity analysis of web pages may not capture subtle yet significant elements that indicate different functionalities, such as hidden buttons or input elements [42].

In contrast, our method only stores the page state representation data and disregards the impact of element position and order. Additionally, we analyzed the extent to which different elements influence page content changes and assigned different weights for three elements for similarity analysis.

7. Conclusions

In this paper, we propose APIMiner, a framework for identifying the APIs of web applications by dynamically traversing web pages based on web page state similarity analysis. When APIMiner accesses a web application, it first builds a web page model based on the HTML elements of the current web page. APIMiner then uses this web page model to represent the state of the page. Then, APIMiner calculates the similarity of each element in the page model and judges the similarity of the page states based on the value of the similarity result of each element. From the different states of the page, APIMiner extracts the data interaction APIs on the page. With this approach, APIMiner traverses as many different web page states as possible, improving the coverage of web application APIs. We conduct extensive experiments to verify the effectiveness of APIMiner and compare it with state-of-the-art similarity analysis methods and tools. The experimental results show that APIMiner achieves good performance in terms of page state similarity analysis, identification of the number of APIs (average of 1136), and code coverage (average of 28,470). In contrast, Enemy of the State, on average, identifies 51 APIs and executes 7578 lines of code, whereas Crawlergo and Wapiti3 identify an average of 171 and 296 APIs and execute 18,060 and 15,293 lines of code, respectively. Compared with these tools, APIMiner identifies an average of 7.96 times more APIs and increases code coverage by an average of 142.72%.

Author Contributions: Conceptualization, Y.C., Y.L. (Yuliang Lu) and Z.P.; Methodology, Y.C., Y.L. (Yuliang Lu), Z.P. and J.C.; Software, Y.C., J.C., Y.L. (Yang Li) and Y.J.; Validation, Y.C., J.C., F.S. and Y.J.; Formal analysis, Y.L. (Yuliang Lu); Investigation, Z.P., J.C., F.S., Y.L. (Yang Li) and Y.J.; Resources, Y.L. (Yang Li) and Y.J.; Writing—original draft, Y.C.; Writing—review & editing, Y.L. (Yuliang Lu) and Z.P.; Visualization, Y.J.; Supervision, F.S.; Funding acquisition, F.S. All authors have read and agreed to the published version of the manuscript.

Funding: This research was funded by the National Key R&D Program of China under Grant 2021YFB3100500.

Data Availability Statement: https://github.com/Cyc1e183/APIMiner, accessed on 25 January 2024.

Conflicts of Interest: Author Yonghui Jiang was employed by the company Anhui Shenwu Information Technology Co., Ltd. The remaining authors declare that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest.

References

1. Grent, H.; Akimov, A.; Aniche, M. Automatically identifying parameter constraints in complex Web APIs: A case study at Adyen. In Proceedings of the 2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP), Madrid, Spain, 25–28 May 2021; pp. 71–80.
2. Atlidakis, V.; Godefroid, P.; Polishchuk, M. Restler: Stateful rest api fuzzing. In Proceedings of the 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE), Montreal, QC, Canada, 25–31 May 2019; pp. 748–758.
3. Hatfield-Dodds, Z.; Dygalo, D. Deriving semantics-aware fuzzers from web API schemas. In Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings, Pittsburgh, PA, USA, 21–29 May 2022; pp. 345–346.
4. Mesbah, A.; Bozdag, E.; Van Deursen, A. Crawling Ajax by inferring user interface state changes. In Proceedings of the 2008 eighth international conference on web engineering, Yorktown Heights, NJ, USA, 14–18 July 2008; pp. 122–134.
5. Mesbah, A.; Van Deursen, A.; Lenselink, S. Crawling Ajax-based web applications through dynamic analysis of user interface state changes. *ACM Trans. Web (TWEB)* **2012**, *6*, 1–30. [CrossRef]
6. Doupé, A.; Cavedon, L.; Kruegel, C.; Vigna, G. Enemy of the state: A state-aware black-box web vulnerability scanner. In Proceedings of the Presented as Part of the 21st {USENIX} Security Symposium ({USENIX} Security 12), Bellevue, WA, USA, 8–10 August 2012; pp. 523–538.
7. Pellegrino, G.; Tschürtz, C.; Bodden, E.; Rossow, C. jäk: Using dynamic analysis to crawl and test modern web applications. In Proceedings of the Research in Attacks, Intrusions, and Defenses: 18th International Symposium, RAID 2015, Kyoto, Japan, 2–4 November 2015; Proceedings 18; Springer: Berlin/Heidelberg, Germany, 2015; pp. 295–316.
8. Eriksson, B.; Pellegrino, G.; Sabelfeld, A. Black widow: Blackbox data-driven web scanning. In Proceedings of the 2021 IEEE Symposium on Security and Privacy (SP), San Francisco, CA, USA, 24–27 May 2021; pp. 1125–1142.

9. Li, X.; Zhang, W.; Wang, D.; Zhang, B.; He, H. Algorithm of web page similarity comparison based on visual block. *Comput. Sci. Inf. Syst.* **2019**, *16*, 815–830. [CrossRef]
10. Wei, Y.; Wang, B.; Liu, Y.; Lv, F. Research on webpage similarity computing technology based on visual blocks. In Proceedings of the Social Media Processing: Third National Conference, SMP 2014, Beijing, China, 1–2 November 2014; Proceedings; Springer: Berlin/Heidelberg, Germany, 2014; pp. 187–197.
11. Gowda, T.; Mattmann, C.A. Clustering web pages based on structure and style similarity (application paper). In Proceedings of the 2016 IEEE 17th International Conference on Information Reuse and Integration (IRI), Pittsburgh, PA, USA, 28–30 July 2016; pp. 175–180.
12. Kang, C.Y. DOM-based web pages to determine the structure of the similarity algorithm. In Proceedings of the 2009 Third International Symposium on Intelligent Information Technology Application, Nanchang, China, 21–22 November 2009; Volume 2, pp. 245–248.
13. What Is REST-REST API Tutorial. 2023. Available online: https://restfulapi.net/ (accessed on 10 November 2023).
14. Pop, D.P.; Altar, A. Designing an MVC model for rapid web application development. *Procedia Eng.* **2014**, *69*, 1172–1179. [CrossRef]
15. Popescu, D.A.; Nicolae, D. Determining the similarity of two web applications using the edit distance. In Proceedings of the Soft Computing Applications: Proceedings of the 6th International Workshop Soft Computing Applications (SOFA 2014), Timisoara, Romania, 24–26 July 2016; Springer: Berlin/Heidelberg, Germany, 2016; Volume 1, pp. 681–690.
16. Vissers, T.; Van Goethem, T.; Joosen, W.; Nikiforakis, N. Maneuvering around clouds: Bypassing cloud-based security providers. In Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, 12–16 October 2015; pp. 1530–1541.
17. Drakonakis, K.; Ioannidis, S.; Polakis, J. ReScan: A Middleware Framework for Realistic and Robust Black-box Web Application Scanning. In Proceedings of the NDSS, San Diego, CA, USA, 27 February–3 March 2023.
18. Crawlergo. 2022. Available online: https://github.com/Qianlitp/crawlergo (accessed on 25 January 2024).
19. Wapiti. 2023. Available online: https://wapiti-scanner.github.io/ (accessed on 25 January 2024).
20. Chen, Y.; Li, Y.; Pan, Z.; Lu, Y.; Chen, J.; Ji, S. URadar: Discovering Unrestricted File Upload Vulnerabilities via Adaptive Dynamic Testing. *IEEE Trans. Inf. Forensics Secur.* **2024**, *19*, 1251–1266. [CrossRef]
21. Najork, M.; Wiener, J.L. Breadth-first crawling yields high-quality pages. In Proceedings of the 10th International Conference on World Wide Web, Hong Kong, China, 1–5 May 2001; pp. 114–118.
22. HTTP Headers-HTTP | MDN. 2020. Available online: https://developer.mozilla.org/en-US/docs/web/http/headers (accessed on 25 January 2024).
23. POST-HTTP | MDN. 2020. Available online: https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods/POST (accessed on 25 January 2024).
24. Alhuzali, A.; Gjomemo, R.; Eshete, B.; Venkatakrishnan, V. {NAVEX}: Precise and scalable exploit generation for dynamic web applications. In Proceedings of the USENIX Security Symposium, Baltimore, MD, USA, 15–17 August 2018; pp. 377–392.
25. W3Techs. W3Techs.com-Usage Statistics and Market Share of Content Management Systems. 2021. Available online: https://w3techs.com/technologies/overview/content_management (accessed on 25 January 2024).
26. Repository Search Results-GitHub. 2022. Available online: https://github.com/search?q=CMS+language%3APHP&type=repositories&s=stars&o=desc&l=PHP (accessed on 25 January 2024).
27. Crawlergo Issues. 2022. Available online: https://github.com/Qianlitp/crawlergo/issues/82 (accessed on 25 January 2024).
28. GitLab. Kostas Drakonakis/rescanApps·GitLab. 2023. Available online: https://gitlab.com/kostasdrk/rescanApps/-/tree/main (accessed on 25 January 2024).
29. Maalej, W.; Robillard, M.P. Patterns of knowledge in API reference documentation. *IEEE Trans. Softw. Eng.* **2013**, *39*, 1264–1282. [CrossRef]
30. Gao, C.; Wei, J.; Zhong, H.; Huang, T. Inferring data contract for web-based API. In Proceedings of the 2014 IEEE International Conference on Web Services, Anchorage, AK, USA, 27 June–2 July 2014; pp. 65–72.
31. Atlidakis, V.; Godefroid, P.; Polishchuk, M. Rest-ler: automatic intelligent rest api fuzzing. *arXiv* **2018**, arXiv:1806.09739.
32. Pandita, R.; Xiao, X.; Zhong, H.; Xie, T.; Oney, S.; Paradkar, A. Inferring method specifications from natural language API descriptions. In Proceedings of the 2012 34th international conference on software engineering (ICSE), Zurich, Switzerland, 2–9 June 2012; pp. 815–825.
33. Kals, S.; Kirda, E.; Kruegel, C.; Jovanovic, N. Secubat: A web vulnerability scanner. In Proceedings of the 15th international conference on World Wide Web, dinburgh, UK, 23–26 May 2006; pp. 247–256.
34. Koppula, H.S.; Leela, K.P.; Agarwal, A.; Chitrapura, K.P.; Garg, S.; Sasturkar, A. Learning url patterns for webpage de-duplication. In Proceedings of the Third ACM International Conference on Web Search and Data Mining, Edinburgh, UK, 23–26 May 2010; pp. 381–390.
35. Nie, T.; Wang, Z.; Kou, Y.; Zhang, R. Crawling result pages for data extraction based on URL classification. In Proceedings of the 2010 Seventh Web Information Systems and Applications Conference, Huhehot, China, 20–22 August 2010; pp. 79–84.
36. Wu, X.; Cao, C.; Wang, Y.; Fu, J.; Wang, S. Extracting knowledge from web tables based on DOM tree similarity. In Proceedings of the Knowledge Science, Engineering and Management: 9th International Conference, KSEM 2016, Passau, Germany, 5–7 October 2016; Proceedings 9; Springer: Berlin/Heidelberg, Germany, 2016; pp. 302–313.

37. Kim, Y.; Park, J.; Kim, T.; Choi, J. Web information extraction by HTML tree edit distance matching. In Proceedings of the 2007 International Conference on Convergence Information Technology (ICCIT 2007), Gwangju, Republic of Korea, 21–23 November 2007; pp. 2455–2460.
38. Abdelnabi, S.; Krombholz, K.; Fritz, M. Visualphishnet: Zero-day phishing website detection by visual similarity. In Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security, Virtually, 9–13 November 2020; pp. 1681–1698.
39. Dalgic, F.C.; Bozkir, A.S.; Aydos, M. Phish-iris: A new approach for vision based brand prediction of phishing web pages via compact visual descriptors. In Proceedings of the 2018 2nd International Symposium on Multidisciplinary Studies and Innovative Technologies (ISMSIT), Ankara, Turkey, 19–21 October 2018; pp. 1–8.
40. Lin, Y.; Liu, R.; Divakaran, D.M.; Ng, J.Y.; Chan, Q.Z.; Lu, Y.; Si, Y.; Zhang, F.; Dong, J.S. Phishpedia: A Hybrid Deep Learning Based Approach to Visually Identify Phishing Webpages. In Proceedings of the USENIX Security Symposium, Boston, MA, USA, 11–13 August 2021; pp. 3793–3810.
41. Liu, R.; Lin, Y.; Yang, X.; Ng, S.H.; Divakaran, D.M.; Dong, J.S. Inferring phishing intention via webpage appearance and dynamics: A deep vision based approach. In Proceedings of the 31st USENIX Security Symposium (USENIX Security 22), Boston, MA, USA, 10–12 August 2022; pp. 1633–1650.
42. Lin, X.; Ilia, P.; Polakis, J. Fill in the blanks: Empirical analysis of the privacy threats of browser form autofill. In Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security, Virtual , 9–13 November 2020; pp. 507–519.