JOURNAL OF INFORMATION SCIENCE AND ENGINEERING 38, 185-205 (2022)
DOI: 10.6688/JISE.202201_38(1).0010

Adaptive Entry Point Discovery for Web Vulnerability Scanning

HSIU-CHUAN HUANG[1,2], ZHI-KAI ZHANG[3], CHUNG-KUAN CHEN[3], WEI-DA HONG[3], JUI-CHIEN JAO[3] AND SHIUHPYNG SHIEH[1]

[1]*Department of Computer Science*
National Yang Ming Chiao Tung University
Hsinchu, 330 Taiwan
[2]*Information and Communication Security Lab*
Chunghwa Telecom Laboratories
Taoyuan, 326 Taiwan
[3]*Department of Computer Science*
National Chiao Tung University
Hsinchu, 330 Taiwan
E-mail: pattyhuang.cs99g@g2.nctu.edu.tw; skyzhang.cs99g@nctu.edu.tw; ckchen@cs.nctu.edu.tw; wdhongtw@gmail.com; {jcjao.cs05g; ssp}@cs.nctu.edu.tw

Entry point collection is crucial to web vulnerability scanning since the collected entry points may contain serious web vulnerabilities such as SQL injection and Cross-Site Scripting (XSS). Most Web Vulnerability Scanners (WVSs) are equipped with crawlers to collect and locate the web pages for testing. The crawlers are intended to discover all links of the web applications being tested. However, exhaustive crawling may not be feasible when time and computation resources are limited, especially for large websites with rapidly and dynamically generated new content. Research studies regarding generic selection policies for web crawlers have been attempted. However, these studies are neither suitable for the search of entry points, nor for WVSs given that their selection policies are intended for content comparison, not for maximizing the test coverage and diversity of functionalities. In this paper, an adaptive entry point crawler named VulCrawl is proposed for WVSs to discover web pages distinct in terms of functionality and code-wise structure. VulCrawl extends the entry point collection and improves WVS code coverage of a target web application. The effectiveness and efficiency of VulCrawl are evaluated using two famous websites. In the experiments, VulCrawl found 2 to 3 times more distinct entry points than those crawled by the web crawler without adopting the adaptive entry point crawling. The results indicate that the proposed selection policy enables web crawling to discover more entry points suitable for WVSs.

Keywords: vulnerability, input validation, crawler, SQL injection, XSS

1. INTRODUCTION

Web applications are widely used in our daily lives. These web applications providing critical services or dealing with sensitive data become attractive targets for cybercriminals. According to Akamai's 2019 state of the Internet report [1], approximately 12 billion web attacks occurred in a 17-month period. To cope with the problem, web security enhancement is an urgent and ineluctable task [2]. Web vulnerability discovery [3] is a way of improving web security by discovering vulnerabilities that exist in web applications in

Received January 2, 2020; revised October 12, 2020; accepted December 25, 2020.
Communicated by Hung-Min Sun.

early stage before cybercriminals take advantage of them.

Black-box testing is a web vulnerability discovery approach which does not rely on the pre-knowledge of the target web applications. During the test, black-box testing attempts to generate malicious inputs automatically and direct the inputs to the target web applications. Since black-box testing does not rely on specific programming languages adopted by target web applications, it can be used to test web applications in a wide variety of instances. Web Vulnerability Scanners (WVSs) are automated tools used to conduct black-box web vulnerability discovery for potential vulnerabilities. To test a web application, a WVS generates malicious web requests and attempts to inject data into the targeted web application. By observing the responses, corresponding vulnerabilities can be discovered.

Most WVSs are equipped with web crawlers to collect web pages from the web applications being tested. A WVS cannot discover the vulnerabilities of a web page unless it can find the web page in the crawling stage. Thus, the coverage of crawling directly affects the test coverage of the WVS. A web application may be large in terms of web page number and be complex owing to a wide variety of information and services [4]. Therefore, the conventional approach, namely *exhaustive crawling*, is no longer a feasible solution to discover web pages of interest when time and computation resources are limited [5]. To cope with the challenge, *selective crawling* is defined as a type of crawling methodology that relies on a selection policy to collect web pages in a web application. The selection policy interferes in the crawling decision so that more web pages of interest can be discovered when time and resource constraints are applied. For WVSs, the interest is to extend the code coverage.

Although *selective crawling* is widely used in discovering web pages that contain content of interest, these crawlers are not suitable for WVSs since the goal of WVSs is to extend the test coverage. To maximize test coverage, diversity of functionalities instead of similarity of content is the main concern. This is because different functionalities usually map to different code segments of the target. As an example, if a link "https://example.com /order.php?user=alice" is collected, a similar link "https://example.com/order.php?user = bob" can be also found. Since the functional purposes of these two web pages are identical, they can share the same code segment. If a WVS allocates resources to analyze this newly encountered web page, it will be a waste. Hence, to extend the test coverage and avoid the waste, *entry point crawling* for WVSs should focus on collecting as many distinct entry points as possible, rather than tracing all links. Our observation from web development experience is that, pages with similar functionality and code-wise structure usually have fewer distinct entry points. To improve the efficiency of gathering distinct entry points, a crawler should assign lower priorities to web pages having similar functional structure. On the other hand, most modern websites generate web pages dynamically, and therefore the crawling strategy should be also dynamically adjusted. *Adaptive crawling* is desirable with crawling strategy and priorities adjusted dynamically. Taking advantage of both *entry point crawling* and *adaptive crawling*, we propose in this paper an *adaptive entry point crawler* named *VulCrawl* for WVSs to discover web pages distinct in terms of functionality and code-wise structure.

There are two main factors that set VulCrawl and generic crawlers apart. First, a generic crawler traces a website and outputs a collection of web pages or links, while VulCrawl outputs a collection of entry points which may receive test payloads. These entry points can be delivered to WVSs and prevent WVSs from repeatedly testing functionally identical web pages. Second, generic crawlers attempt to discover web pages with content-oriented similarities. On the other hand, VulCrawl attempts to discover web pages having different functionality and code-wise structure, which hints function-oriented dissimilarities. With these two factors in mind, VulCrawl is the first attempt to achieve the goal of performing *adaptive entry point crawling*.

Besides designing details, evaluation for the effectiveness and efficiency of VulCrawl is also done in comparison with the web crawler adopting the Breadth-First Search (BFS) selection policy. Two famous websites are selected as the targets. The results show that VulCrawl crawls 2 to 3 times more distinct entry points with acceptable overhead. Contributions of this paper are summarized as follows:

1. A new selection policy is proposed for entry point crawling. This selection policy better reflects the functional and code-wise diversity.
2. An adaptive crawling methodology is proposed to adjust crawling strategy dynamically by utilizing both current crawling status and past crawling experience.
3. A Functional Structure Purifier is also proposed to eliminate unnecessary elements in the Document Object Model (DOM) tree, which reduces the noises and improves the precision during functionality diversity estimation.
4. An adaptive entry point crawler named VulCrawl is proposed, and its evaluations against two famous websites are provided.

The rest of this paper is organized as follows. In Section 2, related work is investigated and problems are introduced. To cope with the problems, a new adaptive entry point crawler is proposed and evaluated in Sections 3 and 4, respectively. The conclusion is given in Section 5.

2. RELATED WORK

Traditionally, at the birth of World Wide Web (WWW), most web pages only consist of a limited number of static documents. Since page analysis is relatively easy, the focus of crawler-related research is to discover desired information according to the contexts and links on the pages. Although conventional crawlers become the core of primitive search engines, they are ineffective against the modern WWW ecosystem since modern web pages are rarely generated by pure HTML.

Later, when server-side scripting techniques, such as ASP, PHP, and other CGI, become popular, web servers have the abilities to handle input queries through HTTP requests, retrieve information from backend databases, and dynamically generate the content of web pages. Since then, web pages, identified by the URLs, may not necessarily correspond to the existing files in web servers, but are related to the information dynamically generated in response to queries and server-side computations. If the crawler only extracts URLs from these dynamically generated pages, most of the web resources linked to the currently analyzed pages may not be reachable. This kind of web is referred to as "deep web". In fact, the deep web dominates the WWW ecosystem nowadays [6, 7]. Thus, many research attempts tried to retrieve deeper pages from web servers, and the crawlers designed for the deep web are called *deep crawlers*.

As the client-side scripting on the page, such as JavaScript, can be involved and executed on the client-side, web pages are not pure static documents and may contain small programs with dynamic behavior. With the help of AJAX, the concept of "web page" becomes blur. It is pretty analogous to a native application because the web page can have state transitions with the execution of scripts. From the user's perspective, it is more like a "web application" instead of a "web page". Hence, this new kind of web is called "Rich Internet Application" (RIA) [8, 9]. To deal with RIAs, the crawlers are referred to as "*RIA crawlers*".

Unlike the aforementioned crawlers which focus on the content or ranks of pages, the crawler proposed in this paper focuses on the functions and the code structure of web pages for web vulnerability discovery. Even though many penetration tools equip built-in crawlers, these crawlers still use the BFS scheduler. Lack of awareness of structure differences makes these crawlers fail to efficiently discover web pages for WVSs, especially against large-scale and dynamic web systems. The gaps can be roughly divided into two issues: resource location and resource selection. The two issues will be elaborated in the next two sections.

2.1 Resource Location

Since client-side web techniques evolve rapidly in recent years, it is mandatory for crawlers to perform dynamic analysis on web pages in order to retrieve as much web content as possible. However, it is a challenging task to dynamically analyze a web page. WAVES [10], proposed by Yao-Wen Huang *et al.*, simulates user events to trigger the behavior of web pages' dynamic components and discovers new links for dynamic content. It adopts a self-learning knowledge base based on a topic model to generate the input data for automated form submissions. FEEDEX [11] by Amin Milani Fard and Ali Mesbah discusses how to achieve better crawling coverage by analyzing event behavior. Due to the flexibility of JavaScript, events can be created, attached to, and detached from web pages during runtime execution. Thus, jÄk [12] by Giancarlo Pellegrino *et al.* focuses on how to monitor all executable events on web pages and precisely locate executable events during dynamic page analysis.

Other studies with additional assumptions on target websites aimed to improve crawling speed. For example, Crawljax [13] by Ali Mesbah *et al.* assumes that the changes of states also lead to the changes of the DOM tree. Therefore, expanding duplicated states can be avoided through detecting duplicated states in the DOM tree. While Crawljax attempts to discover more pages in a web application, VulCrawl makes better resource allocation for discovering diverse, hidden or dynamically generated entry points. On the other hand, An Huiyao *et al.* proposed a strategy called Double Duplicate Elimination Strategy [14]. This strategy identifies the same state based on the assumption that the same XMLHttpRequset (XHR) sequence will lead to the same state. Both studies improve crawling speed but may lose crawling coverage because their assumptions on target websites may not always hold.

These studies have already had great achievements on page analysis. However, most studies use a FIFO queue to record discovered pages without an adaptive resource allocation approach. A common situation on web implementation is that pages with similar functionalities could be clustered together. As a result, if this kind of crawlers is directly applied to web vulnerability discovery, most computation and network resources could be wasted on analyzing pages with similar or even the same code segments. Basically, the codes, instead of the content, are of concern for vulnerability discovery. Although the aforementioned studies may not be suitable for vulnerability discovery, the know-how of locating hidden resources can complement the selection policy mainly proposed in this paper.

2.2 Resource Selection

Exactly as the observation by a survey work from Olston and Najork [6]: "The crawl order is extremely significant because for the purpose of crawling the web can be considered infinite – due to the growth rate of new content, and especially due to dynamically generated content", the order or the selection of candidates to be analyzed is extremely important. Due to dynamically generated pages, fully crawling a large-scale website is usually infeasible. Thus, several selection policies have been proposed for different purposes. These policies are applied to determine the priorities of web pages. According to the purposes, important web pages will have higher priorities. In this way, crawlers can collect as many desirable results as possible from a target website within the limited execution time or resources.

Recent selection policies can be divided into three categories. The first category is for performing topic-focusing search for particular topics. Tianjun Fu *et al.* [15] proposed a selection policy based on sentimental analysis. Their graph-based sentiment (GBS) crawler uses a text classifier to assess the relevance of candidate pages and prioritize more relevant web pages while analyzing web pages. Songhua Xu *et al.* [16] designed a crawler to fulfill the requirement of health research. Their crawler is designed to collect relevant web content with minimal user intervention. Although deep crawlers, such as the aforementioned SmartCrawler [7], may be equipped with selection policies and perform well for content-oriented or topic-focusing analysis, they are not suitable for vulnerability discovery due to the absence of scopes related to the code coverage and lack of security knowledge.

The second category is proposed to efficiently construct a subset of the World Wide Web. Toufik Bennouas and Fabien de Montgolfier [5] proposed a crawler to construct a partial subset of the World Wide Web while preserving similar linking structure and statistical properties of the World Wide Web. This kind of selection policies is helpful to sample the World Wide Web for academic experiments. Still, this category is not suitable for vulnerability discovery because it may preserve redundant page structure when the target website originally has a huge number of redundant pages.

The third category is for PageRank [17] approximation which serves as one of the fundamental algorithms behind search engines. It requires the backend crawlers to perform full crawling of the entire web. Therefore, the corresponding selection policies for this purpose were proposed. Ricardo Baeza-Yates *et al.* [18] provided a comparison between selection policies for PageRank approximation and proposed new strategies as well. This kind of selection policies was designed to collect "important" web content scored by PageRank as fast as possible. Since vulnerability discovery for particular websites has little to do with PageRank, this category is not suitable either.

Even though the aforementioned selection policies provide great contribution to collecting and indexing content-oriented information, these methods are not suitable for vulnerability discovery. Most of the aforementioned work does not deal with the similarity or redundancy of web pages at the functionality level. It is obvious that repeatedly trying to test against the same code segment with the same set of test payloads is a waste of resources. For large-scale web systems, this could make vulnerability discovery not only inefficient but also ineffective. To cope with this problem, a novel crawler will be proposed in the next section.

2.3 Web Vulnerability Scanning

Black-box WVSs simulate attacks against websites under testing and discover vulnerabilities by analyzing their responses. S. Kals developed SecuBat as a generic and modular web vulnerability scanner to analyze websites automatically and discovers exploitable SQL injection and XSS vulnerabilities [19]. Many WVSs have been developed to perform black-box testing on websites for various web vulnerabilities as open source projects or commercial products, such as w3af [20], Arachni [21], Burp Suite [22], Acunetix [23], and WebInspect [24]. The WVSs are based on predefined test payloads, rules and known defects recorded in vulnerability databases. In [3], the test payloads are generated automatically with combinative evasion techniques for WVSs to expand test coverage. The selection policy proposed in this paper can be applied to the built-in crawler of WVSs. For evaluation, the proposed selection policy is tested on Arachni as shown in Section 4.

3. VULCRAWL – AN ADAPTIVE ENTRY POINT CRAWLER

When performing vulnerability discovery, entry point crawling should be first addressed. The web application can be regarded as the aggregation of small programs with their own entry points. A WVS should invoke penetration testing on as many different entry points as possible to improve the test coverage and avoid the waste of resources. The reason is that a WVS cannot reach the code segments containing vulnerabilities unless a relevant entry point is crawled first. Therefore, an effective crawler with the ability to extract different entry points is the first crucial component of a WVS.

3.1 Basic Concepts

Modern web applications become more and more complex. It is common that a large-scale web system may contain millions, sometimes even billions of web pages. These pages are connected with the linking elements, such as hyperlinks, embedded objects, and redirections. Since in most cases these linking elements are in the form of URL, a website can be illustrated as a directed Web Graph *G* as follows.

Web Graph
A web graph $G = (V, E)$ *is a directed graph, where* V *is the set of vertices of* G *and* E *is the set of edges between these vertices. Each vertex represents a web page in the website and is denoted as* v_i*, where* $i \in \mathbb{N}$ *and* $v_i \in V$*. An edge* $e = (v_1, v_2)$ *and* $e \in E$ *iff* v_2 *can be reached from* v_1 *through* e *by triggering some link elements. If* $e = (v_1, v_2)$ *exists in the graph* G*,* v_1 *is said to be a source of* v_2*, and* v_2 *is said to be a target of* v_1*.*

A hyperlink navigation can be triggered by a click event on a hyperlink element in the source web page. Thus, if there are multiple URLs on a web page *P*, all web pages linked by these URLs are considered as the targets of *P*.

VulCrawl outputs a collection of entry points which may receive test payloads. According to RFC 1738, a URL takes the form of "http://<host>:<port>/<path>?<searchpart>" [25], where the <searchpart> stands for the query string in a HTTP request and is recommended to represent in key-value pairs. Each key is a query variable, and values of the key are the values of the query variable. Since these query variables are where attackers can inject attack payloads, besides the path, these variables must be included in the corresponding entry point as well. In addition, the POST, GET parameters and predefined HTTP header fields could be controlled by adversaries. Thus they should be included in entry points. The values of query variables should not be included because requests with different values but the same variables are mostly handled by the same backend script or program. Therefore, for vulnerability discovery, this entry point should be tested only once.

Entry Point

An entry point ep = {Path, QueryVars}, where Path is the resource path in URL format and QueryVars is a set containing all query variables used for this request in URL or message content format.

For example, "http://hostname/user.php?id=2&name=alice" is a web page. Its entry point will be identified as {user.php, (id, name)}. In modern web frameworks, query variables are sometimes embedded in a URL path. Since locating query variables from URLs remains an open problem, a rule-based approach is currently used in VulCrawl. With predefined rules, special placeholders are involved to replace the query variables. Since the query variables can be included in the QueryVars in this way, this kind of URLs can still be processed by VulCrawl for entry point collection.

Even though the concept of an entry point is similar to a web page, there are still differences between them. First, multiple web pages may map only to the same entry point in the aspect of code coverage. For instance, "http://hostname/user.php? id=1&name=alice" and "http://hostname/user.php?id=2&name=bob" are two distinct web pages. However, when performing vulnerability discovery, only one round of tests should be conducted for variables "id" and "name" of "user.php". This is because a WVS generates malicious web requests for "user.php" with crafted values for the variables "id" and "name" during the first run of test and the second round of test will be exactly identical against the same variables. It is a waste if a WVS spends resources testing web pages of the same entry point. On the hand, entry points sometimes may point to the same web page. For example, consider a generic error-handling web page. Many entry points caused by different system failures will be redirected to the same warning web page. Even though the content in the error-handling web page remains unchanged, these entry points are still worth a test for each. Therefore, taking an entry point as the basic testing unit for vulnerability scanning is more precise than taking a web page.

3.2 System Architecture

Before introducing the system architecture of VulCrawl, four states of web pages used in VulCrawl need to be introduced first. During a crawling procedure, web pages are in one of the following four states: *Unknown*, *Discovered*, *Analyzed* and *Expanded*. *Unknown* state indicates the web page is not yet found by a crawler, so the URLs linking to this kind of pages are also unknown. *Discovered* state means the URL linking to the web page is already known through analyzing previous pages, but the actual content of this page has not been fetched yet from the server. Once a web page's content has been fetched, the state is transferred from *Discovered* to *Analyzed*. If the web page does not link to any child page, the state turns from *Analyzed* to *Expanded* directly. If the web page links to one or more child pages, the state turns from *Analyzed* to *Expanded* after all its child page(s) is (are) *Analyzed*.

The system architecture is described in Fig. 1. The component Page Analyzer is responsible for HTML parsing and entry point extraction. Given a web page *p*, Page Analyzer grabs the HTML content of *p* from the web server. After parsing, the corresponding DOM tree is generated, the targets of *p* are discovered and entry points are extracted. Every distinct entry point extracted will be sent to the WVS, and duplicate entry points will be discarded. The component Adaptive Crawler is responsible for choosing next web pages to extract and analyze by utilizing the knowledge constructed during the crawling process. At first, a root web page *r*, which is a URL with or without query variables, is given as a seed page to VulCrawl, where the state of *r* is *Discovered*. *r* is saved into Discovered Webpages Database as the initial state of running, and a new vertex for *r* is added to *G*. Then, Page Analyzer takes a web page *wp* from the Discovered Webpages Database including *r*, and fetches *wp*'s content for analysis. After the analysis, entry points as the outputs of VulCrawl may be found and fed to the WVS for vulnerability discovery. Meanwhile, for any found target web page *cwp*, vertex *cwp* and edge (*wp*, *cwp*) are added to *G*. *cwp* turns to *Discovered* and is saved into Discovered Webpages Database. The state of *wp* transfers from *Discovered* to *Analyzed*. Then, *wp* is saved to Analyzed Webpages Database. Adaptive Crawler then takes the responsibility to choose the most functionally divergent web page(s) from Analyzed Webpages Database for the next round. For each chosen page *nwp*, each found target web page *cnwp* of *nwp* is fed to Page Analyzer for HTML parsing and entry point extraction. Each found target web page *ccnwp* of *cnwp* is saved into Discovered Webpages Database if it is not in *Analyzed* or *Expanded* state. *cnwp* transits to *Analyzed* state, and *nwp* transits to *Expanded* state. Again, Adaptive Crawler starts the next round until all web pages are in *Expanded* state.

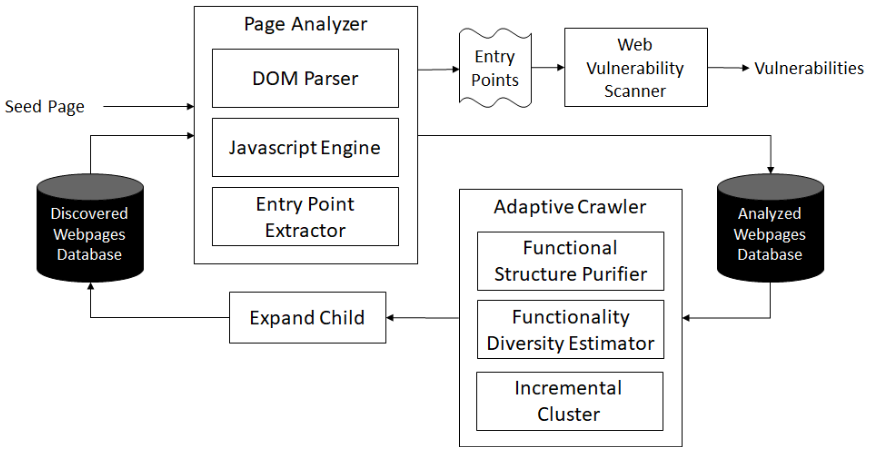


Fig. 1. System architecture.

3.3 Adaptive Crawler

As in Fig. 1, Adaptive Crawler is responsible for choosing the next web pages to extract and analyze. Adaptive crawling should be functionality-oriented instead of content-oriented since the goal is to discover vulnerabilities. Unlike conventional content-oriented crawlers focusing on similarity of page content, a functionality-oriented crawler takes an alternative strategy to analyze web pages according to the diversity of functionalities. Thus, more code segments of a backend program can be tested. Inspired by anomaly detection, this strategy helps WVSs extend the coverage of server-site programs. It is clear that test coverage is critical for penetration testing to ensure the security of target web systems. On the other hand, "adaptive" indicates that the knowledge extracted from crawling history should be maintained to prioritize the candidate pages. An adaptive crawler should use this knowledge to adapt their crawling strategy for more distinct entry points. Due to the fact that information of the target website is unknown while performing black-box testing, the knowledge should be gradually constructed in the crawling process.

Adaptive Crawler contains three modules: Functionality Diversity Estimator, Functional Structure Purifier, and Incremental Clustering. Functionality Diversity Estimator is used to estimate the diversity of web pages, thereby enabling discovery of new entry points. Functional Structure Purifier is designed to extract the functional skeleton via pruning non-functional elements and to further simplify the structure of DOM trees. Incremental Clustering is adopted to reduce the computation cost for Functionality Diversity Estimation. The construction details of Adaptive Crawler are described below.

(A) Functionality Diversity Estimator

Functionality Diversity Estimator estimates the diversity of web pages and helps Adaptive Crawler select web pages having different functionalities and code-wise structures, which hints function-oriented dissimilarities. As described earlier, entry points are the starting point of data flows in a web server. It is reasonable to assume that web pages providing different functionalities contain different entry points pointing to different code segments. Under this assumption, it is intuitive to give higher priority to the page with higher functional diversity when designing a selection policy for entry point crawling. For instance, a web page designed for account setting and a web page designed for product listing usually contain different entry points. Moreover, these entry points can pass different variables and values to different database operations. It is clear that both entry points need input validation testing. As a result, quantifying the diversity of the functionalities between web pages emerges as a critical problem.

Traditional content-based edit distance is not suitable for the measurement of functional diversity because two web pages with identical functional codes may have very different text content. In comparison with content-based heuristics, structural information provides more information about a web page's functionality. Many web pages contain dynamically generated content while having the same functionalities. To provide better user experiences, the layout of these web pages is usually elaborately designed to provide suitable functionality, and the contents are automatically generated after querying the database. For example, two items on the shopping website are represented as two pages with different descriptions and pictures, but their format and functional/scripting codes are the same.

Therefore, the structure of web pages are believed to be more useful for vulnerability discovery than the content.

Knowing that the functionality of a web page is highly related to the structure of the content tree, the measuring method should be based on another kind of edit distance – tree edit distance. As one of derivatives of string edit distance, tree edit distance is an edit distance algorithm that is applied to tree-like data structure. When calculating the tree edit distance from one tree to another, there are three operations: inserting, deleting, and relabeling a node. The basic idea is quite similar to the operations in string edit distance algorithm. Using tree edit distance has another advantage: if the description texts are changed but the functionality remains the same, the distance will be unchanged. As an instance, the web developer may want to update the term “Name:” to “Your Name:”. The distance value will keep unchanged because the tree edit distance algorithm only considers the structural change in the tree.

A special situation may influence the accuracy of using tree edit distance to reflect the functional diversity, that is, longer pages are likely to have longer tree edit distance between each other than the shorter pages. This may result in the bias in issuing higher priority to longer pages. To correct this bias, normalization must be performed. The normalization maps the distance to a value within [0, 1] interval. That means the value of normalized distance will be 0 if two trees are completely identical, and 1 if they are completely different. VulCrawl estimates the diversity of web pages by calculating the normalized tree edit distances of the corresponding DOM trees of web pages. The equation of normalization is as follows, where $|T1|$ represents the number of nodes in the tree $T1$, and similarly for $|T2|$:

$$Distance = \frac{TreeEditDistance(T1, T2)}{|T1| + |T2|}. \tag{1}$$

(B) Functional Structure Purifier

Functional Structure Purifier discovers non-functional elements and remove them as noise. Even though the function structure provides valuable information to estimate the differences of functionality, there exist non-functional elements which make DOM trees different but can be negligible from functionality viewpoint. Therefore, these non-functional elements are considered as noise and can be removed before passing the data to Functionality Diversity Estimator. For instance, text visualization elements such as <b> (a HTML tag for bold text), <br> (a HTML tag for line break), and <u> (a HTML tag for underline) are negligible for web pages functionalities, while elements like <a> (a HTML tag for a link), <script> (a HTML tag for a client-side script), and <object> (a HTML tag for an embedded object) are very important for page functionalities.

Therefore, Functional Structure Purifier is designed to extract the functional skeleton via pruning non-functional elements and to further simplify the structure of DOM tree. Before designing the algorithm of Functional Structure Purifier, non-negligible elements need to be manually classified first since there are a large number of non-functional/negligible elements in all HTML formats. Non-negligible attributes are listed in Table 1 where an attribute may appear in more than one tag. Therefore, focusing on the non-negligible attributes is a more feasible approach. And the algorithm of Functional Structure Purifier is shown in Algorithm 1.

Table 1. Non-negligible attributes of HTML.

Attribute	Related Tag	Usage
action	form	specify where to send the form-data when a form is submitted
cite	blockquote, del, ins, q	specify the URL for a quote document
data	object	specify the URL of the resource to be used by the object
data-*	ALL	store customized data
formaction	button, input	specify where to send the form-data; override "action" attribute
href	a, area, base, link	specify the URL for a page, relative links, external resource, *etc.*
on-*	Almost all tags	specify the event-driven scripts
src	audio, embed, iframe, img, input, script…	specify the URL of a media file or a script file
srcdoc	iframe	specify the in-line HTML content
srcset	img, source	specify the URL of an image
style	ALL	specify the in-line style of an element
value	button, li, option, progress, param	specify the value of an element

```
Algorithm 1: FunctionalStructurePurifier
  Input :
     TreeCurrent: root of a DOM tree before purifying
  Output:
     TreePurified: root of a DOM tree after purifying
1 Function FunctionalStructurePurifier(TreeCurrent):
2    TreePurified ← NULL
3    if TreeCurrent is a leaf node then
        /* remove non-functional leaf node                    */
4       ReserveFlag ← False
5       foreach Attribute of TreeCurrent do
6          if Attribute ∈ ReservedAttributeList then
7             ReserveFlag ← True
8             break
9          end
10      end
11      if ReserveFlag == True then
12         TreePurified ← TreeCurrent
13      else
14         TreePurified ← NULL
15      end
16   else
        /* remove non-functional child                        */
17      RemovedQueue ← NULL
18      foreach Child of TreeCurrent do
19         if FunctionalStructurePurifier(Child) is NULL then
20            RemovedQueue.push(Child)
21         end
22      end
23      TreePurified ← RemoveNode (TreeCurrent, RemovedQueue)
        /* remove non-functional leaf node                    */
24      if TreePurified is a leaf node then
25         TreePurified ←
            FunctionalStructurePurifier(TreePurified)
26      end
27   end
28   return TreePurified
```

(C) Incremental Clustering

In order to identify diverse web pages, the distance of each pair of web pages will be calculated at first. However, as scale of the website becomes larger, the number of distances to be calculated increases in the complexity of $O(n^2)$. Moreover, this procedure will be invoked for many times during the entire crawling. To cope with the problem, the algorithm should be scalable and avoid calculating every distance every time.

VulCrawl utilizes Incremental Clustering as in Algorithm 2 to solve the scalability problem. After separating web pages into a limited number of clusters, representative web pages in each cluster can be selected. Currently, every cluster contains at most x representative pages where x is a configurable parameter in VulCrawl. The x most diverse web pages among the representative pages in this round and newly selected pages are selected as the new representative pages of the cluster for the next round. By calculating only the distance to the representative pages, the number of pairwise comparison can be significantly decreased. In addition, an incremental approach is taken so that only newly incoming web pages are clustered, and recalculating for all web pages can be avoided.

```
Algorithm 2: IncrementalClustering
  Input :
     SelectedPages: set of selected web pages in this round
     TargetPages: set of target web pages in this round
  Output:
     result: True
  Global:
     Clusters: clusters that store all analyzed web pages
     ClusterLevel: n; use the first n folder(s) of Page as the ClusterName
1 Function IncrementalClustering(SelectedPages, TargetPages):
     /* store all new target web pages into clusters        */
2    foreach Page of TargetPages do
3        if Page is placed in the root directory then
4            ClusterName ← RootPlaceHolder
5        else
6            ClusterName ← mapping of first ClusterLevel folder(s) of Page
7        end
8        Add Page to Clusters [ClusterName ]
9        TunePresentativePages(Clusters [ClusterName
           ],SelectedPages,TargetPages)
10   end
11   return True
```

(D) Adaptive Crawling

After describing the system architecture and three modules in previous subsections, the concept "*Adaptive*" involved can be explained. "*Adaptive*" means the aforementioned selection algorithms and policies can be automatically adjusted during the crawling process. This is due to the fact that the components/content of a large-scale and dynamic website often keep changing rapidly. Thus, the priority of each cluster should be adjusted according to both current crawling results and crawling history. In this way, priority of the pages having loops or traps should be lowered, and the resources can be moved to other page clusters.

Since web developers may have their own programing convention, the distribution of distinct entry points may not be uniform. *Potential Rate* is used to denote whether the priority of each cluster should be raised or lowered. In VulCrawl, *Potential Rate* is as follows:

Potential Rate of a Cluster
The "Potential Rate" of cluster i is:

$$Potential_i = \sum_{m=1}^{cur} \in^m r_{i,cur-m} \tag{2}$$

where ε is a configurable decay rate between 0 and 1; $r_{i,j}$ is the number of distinct entry points found in cluster i in round j divided by the total number of distinct entry points found in round j.

The Potential Rates of clusters of each round reflect the found distinct Entry Point distribution for that round. Adaptive Crawler utilizes the Potential Rates of the past crawling process to determine the priority of each cluster in the current round. The priority of resource allocation is implemented by adjusting the number of pages chosen from each cluster. In each round, more pages in a particular cluster will be fetched and analyzed if the cluster has a higher potential rate. Then it uses the aforementioned Functionality Diversity Estimator to determine the most diversity pages. Algorithm 3 shows the algorithm of Adaptive Crawling. The Potential Rate for each cluster is calculated (line 4), then the number of web pages to be fetched in each page cluster can be decided according to these potential rates (lines 5 – 6). The function AdaptiveCrawling provides automatic adaption, which calls the function FindMostDiversePages to perform anomaly detection and select the web pages for next crawling round as shown in Algorithm 4.

As for "Anomaly Detection", the min-max distance algorithm is conducted for every web page in state *Analyzed*. The distances from every page to every cluster's representative page are calculated by Functionality Diversity Estimator (line 6). Among the distances, the smallest one is taken as the diversity score of current page (lines 10-11). Afterward, the

Algorithm 3: Adaptive Crawling

```
  Input :
      cur: current round number
  Output:
      SelectedPages: set of the most diverse web pages in this round
  Global:
      Clusters: clusters that store all analyzed web pages
      NumPagesToSelect: x; select x most diverse pages in this round
      $\epsilon$: decay rate
      $r_{i,j}$: ratio of entry points found in cluster i for round j
1 Function AdaptiveCrawling(cur):
      /* select web pages adaptively for this round      */
2     SelectedPages ← $\phi$
3     foreach $cluster_i$ of Clusters do
4         potential ← $\epsilon \times r_{i,cur-1} + \epsilon^2 \times r_{i,cur-2} + \epsilon^3 \times r_{i,cur-3} + \dots$
5         normalized_potential ← potential × $(1-\epsilon)/\epsilon$
6         SelectedPages ← SelectedPages ∪
            FindMostDiversePages($cluster_i$, $normalized\_potential$ ×
            NumPagesToSelect)
7     end
8     return SelectedPages
```

page with the highest diversity score will be picked out for the next expanding iteration (line 16). That means, the web page farthest from their closest clusters will be chosen. The above algorithm is indeed a special case of *k*th Nearest Neighbor anomaly detection algorithm with *k* = 1. Thus, this algorithm can be extended to *k*th Nearest Neighbor as well as other anomaly detection algorithms if it is needed in the future.

```
Algorithm 4: Finding the most diverse web pages
 Input :
     CurCluster: current cluster
     x: number; select x most diverse pages for this cluster
 Output:
     SelectedPagesForCluster: set of the most diverse web pages for this
     cluster
 Global:
     RPages: set of current representative pages
1  Function FindMostDiversePages(CurCluster, x):
2      RPagesForCluster ← RepresentativePagesForCluster (CurCluster)
3      foreach wp of CurCluster in state Analyzed do
4          $distance_{wp}$ ← -1
5          foreach $wp_r$ of RPagesForCluster do
6              tmp ← FunctionalityDiversityEstimator($wp$, $wp_r$)
               /* calculate the smallest distance                    */
7              if $distance_{wp}$ == -1 then
8                  $distance_{wp}$ ← tmp
9              else
10                 if tmp is less than $distance_{wp}$ then
11                     $distance_{wp}$ ← tmp
12                 end
13             end
14         end
15     end
16     SelectedPagesForCluster ← x wp with the top high $distance_{wp}$
17     return SelectedPagesForCluster;
```

4. EVALUATION OF VULCRAWL

The evaluation of VulCrawl contains three parts: effectiveness evaluation, efficiency evaluation and case study. The details of above three evaluation parts are in the following three subsections.

4.1 Effectiveness Evaluation

The main goal of VulCrawl is adaptively finding more distinct entry points to expand the test coverage against large-scale websites. This effectiveness evaluation is to prove that VulCrawl indeed achieves the goal. In this evaluation, two crawlers were used. The first crawler was the original crawler built in Arachni, and the other was VulCrawl with different configurations.

For convincingness, two different real famous dynamic and large-scale websites, Website-1 and Website-2, are selected as the target websites for evaluation. Website-1 is a website that provide online news, music, video, *etc.* Website-2 is an online auction and shopping website. Both websites consist of a massive number of web pages. Because the pages could be generated dynamically all the time, it is infeasible to perform exhaustive crawling. Thus, the value of VulCrawl can be shown in this kind of situation. To avoid any possible impacts on the network or the performance of target websites, only normal HTTP/HTTPS requests were sent out and a proper time interval was involved.

(A) The Number of Distinct Entry Points Found in a Period of Time

As shown in Fig. 2, this evaluation contains six different configurations. The first one is the original crawler in Arachni which is labeled as "FIFO" since Arachni uses a FIFO queue. The others are VulCrawl with five different decay rates (approximated to 0, 0.25, 0.5, 0.75 and 1), which indicate the importance of past experience. From Fig. 2, VulCrawl behaves better both in the Website-1 and Website-2 cases. The results show that VulCrawl crawls 2 to 3 times more distinct entry points than those crawled by the original crawler in Arachni. Note that our approach is not dedicated to Arachni. It can be also applied to other crawlers to discover more entry points.

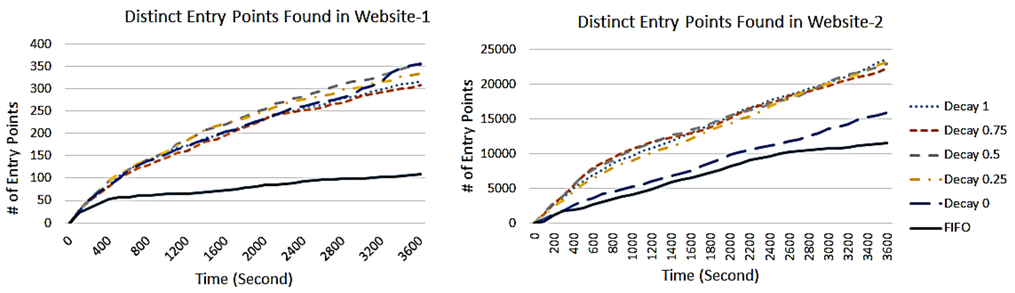


Fig. 2. Found distinct entry points.

(B) Entry Points Crawling Per Iteration

The second evaluation has the same six configurations as the first one and is shown in Fig. 3. This evaluation shows that, the web pages selected by VulCrawl contain multiple times of distinct entry points on average. This result indicates that focusing on the functional diversity of web pages does help to find more distinct entry points. By assumption, this leads to extending the test coverage.

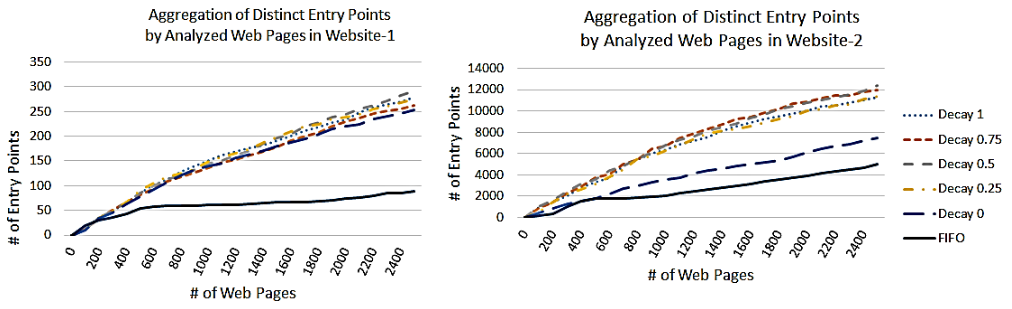


Fig. 3. Aggregation of entry points versus web page number.

Another interesting result is shown in Fig. 4. More entry points were found in first few iterations of 100 web pages than the later iterations. The trend of decrease is reasonable because the ratio of duplicated entry points increased with the growth of analyzed web pages. Suddenly increasing jumps may exist. The jumps indicate that web pages with many diverse child pages are found. The suddenly jumps happened more frequently when the decay rate is approximated to "0". This is because considering only current crawl status could result in instability. Hence, the other decay rates are believed to be more stable and suitable for web vulnerability scanning.

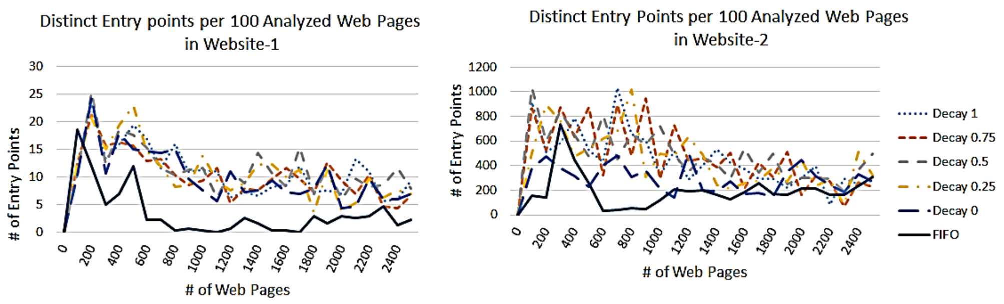


Fig. 4. Entry points per 100 pages.

(C) Decay Rate Tuning

In this experiment, the goal is to find which decay rate is better for Adaptive Crawler. The priority of each cluster is calculated by two factors: the current number and the historic numbers of entry point found. The argument ε is used to assign the decay rate between current and previous iterations. That is, if the ε is larger, the performance of historic iterations is more considerable, and vice versa.

This experiment contains five different values of ε. The result is shown in Fig. 3, and "$\varepsilon = 0.5$" performs well against Website-1 and Website-2. When the decay rate is approximated to 0, VulCrawl works well in Website-1, but not so significant in Website-2.

4.2 Efficiency Evaluation

Besides the effectiveness evaluation, efficiency evaluation is also performed. The overhead is evaluated by comparing the number of analyzed pages within the same time period. The target websites are still Website-1 and Website-2, and the result is shown in Fig. 5. For Website-1, FIFO analyzed 3,573 pages in 3,600 seconds while VulCrawl with 0.5 decay rate analyzed 3,476 pages. For Website-2, FIFO analyzed 5,933 pages in 3,600 seconds while VulCrawl with 0.5 decay rate analyzed 5,536 pages. The statistics above showed that the overhead is around 3%-7%. Since the number of distinct entry points found by VulCrawl is more than twice as FIFO, this overhead is acceptable.

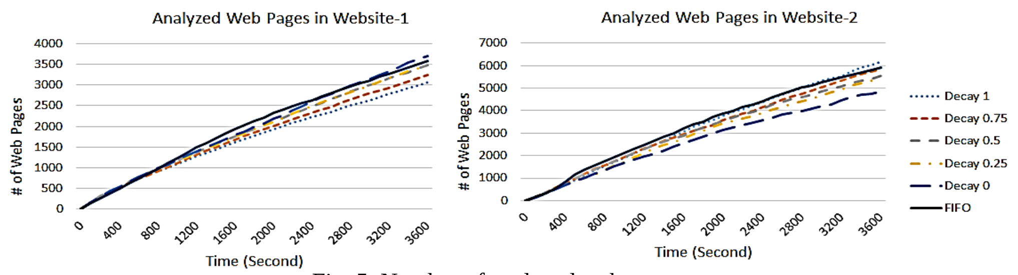


Fig. 5. Number of analyzed web pages.

4.3 Case Study

In order to prove that VulCrawl can be applied to real world applications, two popular frameworks including WordPress and SchoolMate are employed as case studies. To demonstrate that VulCrawl can discover entry points of web applications that contain vulnerabilities automatically, known-vulnerable versions of WordPress and SchoolMate are used in the experiments. We use VulCrawl to discover entry points of the target web applications and use an existing open-source WVS module to perform vulnerability testing.

For WordPress being test, three SQL injections (SQLI), two XSSs and one directory traversal (DirTrav) vulnerabilities were found. For SchoolMate being test, one SQL injection and one XSS vulnerabilities were found. The discovered vulnerabilities are listed in Table 2. The last two columns of Table 2 show the vulnerability type found and the test result, respectively. The test result "Fully Reproduced" means both the entry point and attack payload were successfully found. The test result "Discovered" means the entry point was found by VulCrawl, but the WVS module failed to generate the right payload. This situation indicates the immaturity of the WVS. Even in this case, the goal of VulCrawl was achieved to discover entry points. The experiment results show that VulCrawl indeed helps a WVS discover the entry points linking to vulnerabilities automatically.

Table 2. Discovered vulnerabilities.

CVE	Framework	Entry Point	Type	Result
CVE-2007-4894	WordPress 2.0.6	xmlrpc.php, {post_body}	SQLI	Discovered
CVE-2007-6318	WordPress 2.0.6	wp-includes/query.php, {s}	SQLI	Discovered
CVE-2015-4064	WordPress 2.0.6	wp-admin.php/post.php, {post}	SQLI	Fully Reproduced
CVE-2010-5295	WordPress 2.0.6	wp-admin/plugins.php, {plugin}	XSS	Fully Reproduced
CVE-2016-1564	WordPress 2.0.6	wp-admin/themes.php, {template}	XSS	Fully Reproduced
CVE-2008-4769	WordPress 2.0.6	wp-admin/themes.php, {?}	DirTrav	Discovered
N/A	SchoolMate	schoolmate/index.php, {sitetext}	SQLI	Fully Reproduced
N/A	SchoolMate	schoolmate/index.php, {semester}	XSS	Fully Reproduced

5. SUMMARY

An effective crawler with the ability to collect the web pages and extract the entry points for testing is a crucial component of a WVS. A WVS cannot discover the vulnerabilities of a web page unless it can find the web page in the crawling stage. However, exhaustive crawling may not be feasible when time and computation resources are limited, especially for large websites with rapidly and dynamically generated new content. To extend the test coverage and avoid the waste, *entry point crawling* for WVSs should focus on collecting as many distinct entry points as possible, rather than tracing all links. Furthermore, *Adaptive crawling* is desirable with crawling strategy and priorities adjusted dynamically. Although some selection policies have been proposed for web crawlers, they are not suitable for WVSs given that their selection policies are intended for content comparison. To maximize test coverage, a web crawler of a WVS should discover web pages having different functionalities. In this paper, an adaptive entry point crawler named VulCrawl is proposed for WVSs to discover web pages distinct in terms of functionality and code-wise structure. VulCrawl utilizes the structure discrepancy and past experience to discover more distinct web pages and entry points for vulnerability scanning. It proposes a new selection policy. It estimates the diversity of web pages by calculating the normalized tree edit distances of the corresponding DOM trees of web pages, and selects the most diverse pages for discovery of new entry points. This selection policy better reflects the functional and code-wise diversity. To reduce the noises and improve the precision during functionality diversity estimation for web pages, a module Functional Structure Purifier is designed to extract the functional skeleton via pruning non-functional elements and to further simplify the structure of DOM trees. VulCrawl adopts an adaptive crawling methodology to adjust crawling strategy dynamically by utilizing both current crawling status and past crawling experience so that more web pages of different functionalities are discovered.

Evaluation for the effectiveness and efficiency of VulCrawl is done in comparison with the original crawler, which does not adopt the adaptive entry point crawling, built in Arachni. Two different famous dynamic and large-scale websites, Website-1 and Website-2, are selected as the target websites for evaluation. VulCrawl are experimented with five different decay rates (approximated to 0, 0.25, 0.5, 0.75 and 1), which indicate the importance of past experience. In the experiments, the results show that VulCrawl crawls 2 to 3 times more distinct entry points than those crawled with the original crawler. The results also show that the web pages selected by VulCrawl contain more distinct entry points, which indicates that focusing on the functional diversity of web pages does help to find more distinct entry points. Five different decay rates are experimented. When the decay rate is set to approximated to 0, VulCrawl works well in Website-1, but not so significant in Website-2. This is because VulCrawl with a decay rate approximated to 0 considers only the current crawling status. It does not consider the past crawling experience. VulCrawl with a decay rate not approximated to 0 behaves well both in Website-1 and Website-2. A decay rate not approximated to 0 is more stable and suitable for web vulnerability scanning. The overhead is evaluated by comparing the number of analyzed pages within the same time period. The statistics show that the overhead is around 3%-7%. Since the number of distinct entry points found by VulCrawl is more than twice as that by the original crawler, this overhead is acceptable. Case studies on WordPress and SchoolMate are also conducted to demonstrate that VulCrawl can assist Web Vulnerability Scanners in discovering more vulnerabilities. The results indicate that VulCrawl enables web crawling to discover more entry points suitable for WVSs.

REFERENCES

1. Akamai, "Akamai's [state of the internet]/security: Web attacks and gaming abuse," 2019, Vol. 5, https://www.akamai.com/us/en/multimedia/documents/state-of-the-internet/soti-security-web-attacks-and-gaming-abuse-report-2019.pdf.
2. C. K. Chen, Z. K. Zhang, S. H. Lee, and S. Shieh, "Penetration test in the IoT age," *IEEE Computer*, Vol. 51, 2018, pp. 82-85.
3. H. C. Huang, Z. K. Zhang, H. W. Cheng, and S. W. Shieh, "Web application security: attacks, countermeasure, and pitfalls," *IEEE Computer*, Vol. 50, 2017, pp. 81-85.
4. Dan, "How big is a large website ?" 2011, http://contentini.com/how-big-is-a-large-website-planning-the-content-audit-app/.

5. T. Bennouas and F. de Montgolfier, "Random web crawls," in *Proceedings of the 16th International Conference on World Wide Web*, 2007, pp. 451-460.
6. C. Olston and M. Najork, "Web crawling," *Foundations and Trends in Information Retrieval*, Vol. 4, 2010, pp. 175-246.
7. F. Zhao, J. Zhou, C. Nie, H. Huang, and H. Jin, "Smartcrawler: a two-stage crawler for efficiently harvesting deep-web interfaces," *IEEE Transactions on Services Computing*, vol. 9, 2016, pp. 608-620.
8. S. M. Mirtaheri, M. E. Dinc¸türk, S. Hooshmand, G. V. Bochmann, G. V. Jourdan, and I. V. Onut, "A brief history of web crawlers," in *Proceedings of Conference of the Center for Advanced Studies on Collaborative Research*, 2013, pp. 40-54.
9. S. Gupta and K. K. Bhatia, "A comparative study of hidden web crawlers," *arXiv Preprint*, 2014, arXiv:1407.5732.
10. Y. W. Huang, S. K. Huang, T. P. Lin, and C. H. Tsai, "Web application security assessment by fault injection and behavior monitoring," in *Proceedings of the 12th International Conference on World Wide Web*, 2003, pp. 148-159.
11. A. M. Fard and A. Mesbah, "Feedback-directed exploration of web applications to derive test models." in *Proceedings of the 24th IEEE International Symposium on Software Reliability Engineering*, Vol. 13, 2013, pp. 278-287.
12. G. Pellegrino, C. Tschürtz, E. Bodden, and C. Rossow, "jÄk: using dynamic analysis to crawl and test modern web applications," in *Proceedings of International Symposium on Recent Advances in Intrusion Detection*, 2015, pp. 295-316.
13. A. Mesbah, A. Van Deursen, and S. Lenselink, "Crawling ajax-based web applications through dynamic analysis of user interface state changes," *ACM Transactions on the Web*, Vol. 6, 2012, pp. 1-30.
14. A. Huiyao, S. Yang, Y. Tao, L. Hui, Z. Peng, and Z. Jun, "A new architecture of ajax web application security crawler with finite-state machine," in *Proceedings of International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery*, 2014, pp. 112-117.
15. T. Fu, A. Abbasi, D. Zeng, and H. Chen, "Sentimental spidering: leveraging opinion information in focused crawlers," *ACM Transactions on Information Systems*, Vol. 30, 2012, pp. 1-30.
16. S. Xu, H. J. Yoon, and G. Tourassi, "A user-oriented web crawler for selectively acquiring online content in e-health research," *Bioinformatics*, Vol. 30, 2013, pp. 104-114.
17. L. Page, S. Brin, R. Motwani, and T. Winograd, "The PageRank citation ranking: Bringing order to the web," Technical Report No. SIDL-WP-1999-0120, Stanford InfoLab, 1999.
18. R. Baeza-Yates, C. Castillo, M. Marin, and A. Rodriguez, "Crawling a country: better strategies than breadth-first for web page ordering," in *Special interest tracks and posters of the 14th International Conference on World Wide Web*, 2005, pp. 864-872.
19. S. Kals, E. Kirda, C. Kruegel, and N. Jovanovic, "Secubat: a web vulnerability scanner," in *Proceedings of the 15th International Conference on World Wide Web*, 2006, pp. 247-256.
20. A. Riancho, "w3af – Web application attack and audit framework," https://github.com/andresriancho/w3af, 2015.

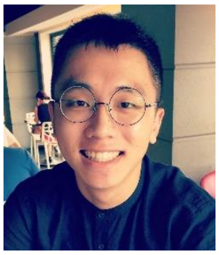
21. Arachni, "Arachni – web application security scanner framework," http://www.arachni-scanner.com/, 2018.
22. PortSwigger, "Burp Suite – application security testing software," https://portswigger.net/burp, 2020.
23. Acunetix, "Acunetix web vulnerbility scanner," https://www.acunetix.com/, 2020.
24. Micro Focus, "WebInspect," https://www.microfocus.com/zh-tw/products/webinspect-dynamic-analysis-dast/overview, 2020.
25. T. Berners-Lee, L. Masinter, and M. McCahill, "Uniform resource locators (URL)," https://tools.ietf.org/html/rfc1738, 1994.

Hsiu-Chuan Huang (黃秀娟) received her M.S. degree in Computer Science and Information Engineering from National Taiwan University, Taiwan. She is currently a Ph.D. student in the Department of Computer Science at National Yang Ming Chiao Tung University, Taiwan, and a Security Researcher of Chunghwa Telecom Laboratories. Her current research interests include web security, network security and machine learning.

Zhi-Kai Zhang (張智凱) received his Ph.D. degree in Computer Science from National Chiao Tung University in 2018. He had led Teaching and Learning Development Division of Hacker College of NCTU. He is currently leading an R&D team for a startup, and is also a course lecturer of Hacker College. His research interest includes cryptography, cloud security, IoT security, penetration testing, and information security education.

Chung-Kuan Chen (陳仲寬) is currently a Senior Researcher in CyCraft, and is responsible for organizing their research team. He earned his Ph.D. degree of Computer Science and Engineering from National Chiao Tung University. His research focuses on network attack and defense, machine learning, software vulnerability, malware and program analysis. He also dedicates to security education. Founding of NCTU hacker research club, he trains students to participate world-class security contests, and has experience of participating DEFCON Final CTF. He organized the research clubs to join some bug bounty projects and discovered some CVEs in COTS software and several vulnerabilities in campus websites.

Wei-Da Hong (洪偉達) is an Engineer from Taiwan. He loves Python and tries to bring the Zen of Python into other programming languages. With the advice from his mentors, he received his M.S. degree in Computer Science from National Chiao Tung University in 2017.



Jui-Chien Jao (饒瑞謙) received his M.S. degree in Computer Science from National Chiao Tung University, Taiwan. His research interests include web security, and penetration testing. He is currently a member of SIRT in Synology Inc.

Shiuhpyng Shieh (謝續平) received his Ph.D. degree in Electrical and Computer Engineering from the University of Maryland, College Park, and is currently a Chair Professor of Computer Science Department of National Yang Ming Chiao Tung University. He is an IEEE Fellow and ACM Distinguished Scientist. His research interests include enterprise security, intrusion detection, threat hunting, and user behavior analytics using AI.