# EvoCrawl: Exploring Web Application Code and State using Evolutionary Search

Xiangyu Guo
University of Toronto
xiangyu.guo@mail.utoronto.ca

Akshay Kawlay
University of Toronto
ak.kawlay@mail.utoronto.ca

Eric Liu
University of Toronto
ec.liu@mail.utoronto.ca

David Lie
University of Toronto
david.lie@utoronto.ca

***Abstract*—As more critical services move onto the web, it has become increasingly important to detect and address vulnerabilities in web applications. These vulnerabilities only occur under specific conditions: when 1) the vulnerable code is executed and 2) the web application is in the required state. If the application is not in the required state, then even if the vulnerable code is executed, the vulnerability may not be triggered. Previous work naively explores the application state by filling every field and triggering every JavaScript event before submitting HTML forms. However, this simplistic approach can fail to satisfy constraints between the web page elements, as well as input format constraints. To address this, we present EvoCrawl, a web crawler that uses evolutionary search to efficiently find different sequences of web interactions. EvoCrawl finds sequences that can successfully submit inputs to web applications and thus explore more code and server-side states than previous approaches. To assess the benefits of EvoCrawl, we evaluate it against three state-of-the-art vulnerability scanners on ten web applications. We find that EvoCrawl achieves better code coverage due to its ability to execute code that can only be executed when the application is in a particular state. On average, EvoCrawl achieves a 59% increase in code coverage and successfully submits HTML forms 5× more frequently than the next best tool. By integrating IDOR and XSS vulnerability scanners, we used EvoCrawl to find eight zero-day IDOR and XSS vulnerabilities in WordPress, HotCRP, Kanboard, ImpressCMS, and GitLab.**

## I. Introduction

Since 2017, broken access control and XSS code injection have been consistently ranked among the most prevalent vulnerabilities in OWASP Top 10. As stated in OWASP's 2021 report [1], 94% of tested applications exhibited some form of broken access control or injection vulnerabilities, underscoring the need for developers to safeguard their web applications against these defects. There are two main approaches to detecting such vulnerabilities: static analysis and dynamic analysis. Static analysis tools [2], [3], [4], [5] require the application's source code, which limits their applicability to applications written in other programming languages. Conversely, dynamic analysis can be agnostic to the programming language, but can only detect vulnerabilities if they occur during the tool's application exploration. Therefore, dynamic analysis focuses on exploring as much code as possible.

Network and Distributed System Security (NDSS) Symposium 2025
23–28 February 2025, San Diego, CA, USA
ISBN 979-8-9894372-8-3
https://dx.doi.org/10.14722/ndss.2025.23366
www.ndss-symposium.org

Web vulnerability scanners are often paired with a web crawler, which attempts to maximize the code coverage of an application by scanning as many pages as possible. However, simply crawling pages is not sufficient to maximize code coverage. This is because some code in a web application is associated with functionality that can only be triggered when the application is in a specific server-side state. For instance, in GitLab, a web-based revision control system, functions that enable users to manipulate code repositories require a repository to be created first. As a result, a vulnerability scanner will be unable to test any of that code if it is unable to interact with GitLab and create new repositories. Therefore, to achieve good code coverage, and thus find more vulnerabilities, a web crawler must explore both web pages *and* application states.

This importance has not been lost in previous approaches to web application vulnerability detection. For example, BlackWidow [6] and Enemy of the State [7] incorporate HTML forms into their navigation graph, and submitting these HTML forms enables them to explore different server-side states. In addition, to handle AJAX-enabled dynamic web pages, BlackWidow also adds JavaScript events to its navigation graph. Triggering JavaScript events can enable additional fields and elements on a web page, allowing BlackWidow to submit more data and explore more server-side states.

However, to correctly submit data to a web application that will modify the server-side state, a web crawler must satisfy both ordering and formatting constraints on interactions with HTML elements and trigger JavaScript events in the right order. For example, to submit data via a form, the web application may impose an ordering constraint that requires the crawler to first enter data into text fields, or select the correct options from dropdown boxes or radio buttons, before hitting the submit button. Similarly, a web application may impose formatting constraints such that fields that require a date or an e-mail must have well-formed inputs. Finding a sequence that meets ordering constraints requires a search over all possible sequences of interactions with HTML elements and JavaScript events (which we collectively refer to as web elements), which grows exponentially with the number of such elements and events. BlackWidow and Enemy of the State naively avoid searching this large space by filling in every input field in HTML forms, and only enumerating all sequences of JavaScript events and HTML forms. While this reduces the search space, it increases the chances that they will violate formatting constraints, as some web pages contain optional fields, which could have been left blank.

To address this, we propose EvoCrawl, which overcomes both ordering and formatting constraints, enabling it to submit more data and explore more server-side states than previous approaches. EvoCrawl achieves this by performing a *fine-grain* search of sequences of interactions with web elements, including individual HTML fields. Searching subsets of HTML fields enables EvoCrawl to submit a larger diversity of inputs to the web application and generate a larger diversity of server states. In addition, filling some fields and leaving others blank enables EvoCrawl to successfully submit data in cases where EvoCrawl finds a field's formatting constraints are too difficult for EvoCrawl to infer, but which also happens to be optional. The drawback with this approach is that a fine-grain search results in a larger search space, which EvoCrawl addresses with two key innovations. First, EvoCrawl uses an evolutionary algorithm with a fitness function that enables it to focus its search on sequences that are able to successfully submit inputs, or reveal more web elements to interact with. Second, to further reduce the search space, EvoCrawl detects dependencies between web elements, enabling it to eliminate sequences that violate these dependencies.

To measure the improvements these two techniques confer, we evaluate EvoCrawl on 10 modern web applications and compare its performance against three modern black-box scanners: BlackWidow [6], JAK [8], and CrawlJAX [9]. EvoCrawl can also be combined with various detector modules to detect different types of vulnerabilities. We have implemented IDOR (Insecure Direct Object Reference) and XSS (Cross-Site Scripting) vulnerability detectors in EvoCrawl. The IDOR vulnerability detector (IVD) not only automatically categorizes resources but also exhibits a low rate of false positives when detecting IDOR vulnerabilities. Inspired by BlackWidow [6], the XSS vulnerability detector (XVD) injects XSS payloads containing unique integers into every feasible input field. By monitoring and tracking these integers, the XVD can identify the relationships between input sources and sinks, subsequently exposing the payloads. These detectors enable EvoCrawl to find 8 zero-day IDOR and XSS vulnerabilities. We have responsibly disclosed all vulnerabilities and the developers have either fixed or acknowledged all except two of them.

The following are our main research contributions:

- We identify that successfully and efficiently executing client-side events is a significant impediment to web application exploration, which should be overcome for web vulnerability scanners to increase code coverage and find vulnerabilities.
- We present EvoCrawl, which combines an evolutionary search algorithm with standard crawling to more comprehensively explore web application code to find vulnerabilities. EvoCrawl is openly available at https://github.com/dlgroupuoft/evocrawl
- We integrate all the detector modules into EvoCrawl and evaluate its performance by comparing it against 3 modern scanners on 10 web applications. EvoCrawl successfully identifies eight zero-day bugs in WordPress, HotCRP, Kanboard, ImpressCMS, and Gitlab. It achieves an average code coverage increase of 59% and outperforms BlackWidow by submitting HTML forms with the POST method 5 times more frequently.

This paper is structured as follows: Section III offers a description of EvoCrawl's design, while Section IV elaborates on its implementation details. Section V compares EvoCrawl on the metrics of code coverage and the ability to submit HTML form inputs to an application against other state-of-the-art web vulnerability scanners. We then evaluate the vulnerability detection ability of EvoCrawl and detail the new vulnerabilities found in Section VI, followed by Sections VII and VIII, outlining the limitations of EvoCrawl and reviewing related works in the field. Finally, we draw conclusions in Section IX.

## II. Motivation

As mentioned in Section I, modern web pages heavily use JavaScript to asynchronously update and modify web pages. The crawler needs to trigger certain JavaScript events to make additional web elements or links accessible. Therefore, BlackWidow [6] incorporates JavaScript events into its navigation graph. By enumerating all possible sequences of JavaScript events and HTML forms, BlackWidow aims to find the sequences that satisfy the dependencies or constraints imposed by the web pages. However, as the numbers of JavaScript events and HTML forms increases, the search space expands exponentially. Even a small web page with just 10 JavaScript events and 1 HTML form would generate a search space of $n^{11}$ for sequences of $n$ interactions, which can quickly become too large to search even for modest values of $n$.

A sequence here refers to some JavaScript events and/or HTML forms placed in a certain order, and the targeted sequences are the ones that can reveal new links or explore server-side states after their executions. To find targeted sequences inside this vast search space, we need the crawler to accomplish two steps: 1) reduce the search space and 2) search for the targeted sequences efficiently. Certain web elements including ones that listen to the JavaScript events are dependent on each other. Specifically, certain web elements are not visible or active unless the user interacts with other web elements first. For instance, the form in Fig. 1 is only visible and active after interacting with the arrow button in the red box that can expand the component, or specific elements may become visible only following certain prerequisite interactions. By enforcing the dependency information on the order of sequences, EvoCrawl avoids trying impossible sequences, thus reducing the search space.

EvoCrawl can accomplish the first step by tracking and enforcing dependencies. Then, to search the remaining space

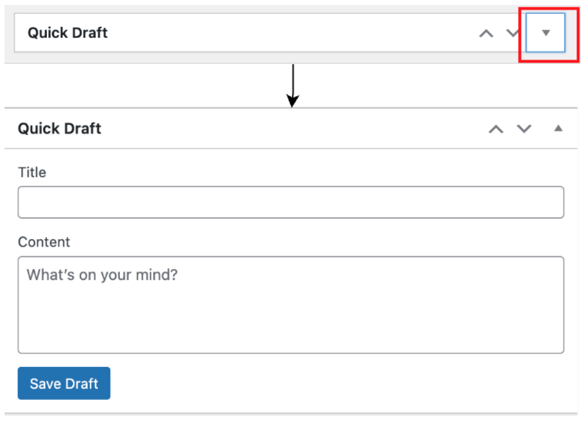


Fig. 1: An Illustrative Example for WordPress

efficiently, we propose the use of an evolutionary genetic search algorithm, which is a good fit for this problem. First, its crossover operation allows the crawler to generate variations of sequences based on previously successful sequences. Second, with feedback from the web browser and database, the evolutionary search can use a fitness function to identify the sequences more likely to reveal new links and explore application states.

Additionally, unlike BlackWidow, which attempts to fill in all input fields to submit a form, EvoCrawl searches for various sequences of interactions for form submission. This variation in approach is necessary because some input fields, such as those requiring very specific formats like“YEAR-MONTH-DAY,” cannot have their values easily inferred by heuristics. For forms containing these complex input fields, EvoCrawl can find different sequences for submission. These sequences may include filling in all fields or omitting some of the fields. The intuition is that while EvoCrawl may not generate the correct values for complex input fields, it can still find the sequence that successfully submits the form by bypassing these fields. For instance, in a user registration process on a website, fields like “username”, “password”, and “email” must be completed, whereas others like“birthday” or“time zone” are optional and can be left blank. However, for non-optional fields requiring inputs that heuristics cannot infer, EvoCrawl still fails to submit the related form.

## III. Design

We integrate the dependency tracking mechanism and the genetic algorithm into the Evolutionary Search Module (ESM) of EvoCrawl. For each web page, the ESM evolves through several generations. In each generation, it begins by generating sequences of web element interactions, which involve three different operations: crossover, mutation, and random combinations. The crossover operation allows the ESM to concatenate two sequences together so the generated sequences can inherit the properties of the previous sequences. The ESM employs the mutation operation to enforce dependency tracking within the sequences and utilizes random combinations to introduce diversity into the generated sequences.

Following the sequence generation, the ESM executes these sequences via the User interface (UI) in the browser and evaluates them using a fitness function. During the execution of each sequence, the fitness function assigns a score to the sequence based on feedback from the server-side database and the client-side browser. The fitness function indicates how “good” the sequence is in achieving its objectives.

Additionally, during the execution of each sequence, the crawler constructs and updates a dynamic map based on its observed dependency information. Within this map, each interaction is associated with elements that only become visible after the crawler executes the interaction. Subsequently, the crawler introduces mutations to the sequence to enforce the constraints outlined in the dynamic map

EvoCrawl’s evolutionary search algorithm focuses on searching for sequences of interactions on a page in a web application. However, to be effective, it needs to be run on as many pages in the web application as possible. This is partially achieved by taking new URLs it discovers during its search and storing them as targets for later search sessions. However, the rate at which it discovers these new pages is affected by the large search space of application web pages. As a result, EvoCrawl is actually composed of two modules that perform two types of searches. The Evolutionary Search Module (ESM) performs the aforementioned evolutionary search of interaction sequences, while a Page Collection Module (PM) interacts with each web element only once to rapidly collect different links.

Figure 2 illustrates the architecture of EvoCrawl. Throughout the scanning process, both the PM and ESM exchange pages they have found: the ESM uses these as target pages for its evolutionary search, and the PM uses these as starting points to crawl for more pages and other web elements. Both modules exchange URLs with the two vulnerability detectors: the IDOR vulnerability detector (IVD) and the XSS vulnerability detector (XVD). The IVD classifies the URLs and assesses the access control level of the private ones. The XSS vulnerability detector generates JavaScript payloads, which are injected by both PM and the ESM into the web application (sources), and also monitors for the successful execution of these payloads (sinks). We describe each of these components in more detail below.

### A. Page Collection

The objective of the Page Collection Module (PM) is to find URLs that can be passed to the Evolutionary Search Module (ESM). To do this, it recursively searches the web application by identifying web elements and interacting with them. Each interaction that triggers DOM changes is called an $EVENT$, and is associated with the URL of the page on which it is found to form a tuple $< URL, EVENT >$, which we call a *seed*. To perform its search, the PM stores seeds in a queue, with which it iteratively performs a three-stage crawling process on

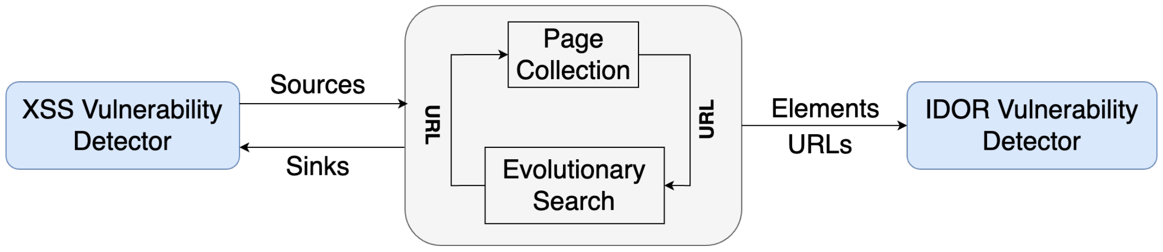


Fig. 2: Block Diagram of EvoCrawl

each seed, which consists of links crawling, events crawling, and forms crawling.

During the links crawling stage, the PM performs the following steps. First, it executes the seed by navigating to the URL specified in the current seed. If the value of EVENT is not empty, the associated DOM event is triggered. Once the seed execution is successful, the crawler extracts all `href` values from the anchor elements on the web page. These `href` values are then used to construct new seeds, which will be added to the end of the queue if they are not already in the queue or have not been visited before.

During the event crawling stage, the PM interacts with each interactable element on the web page to determine if the element can invoke a DOM event. If, after each interaction, the DOM changes without refreshing the page or navigating to other pages, the crawler identifies the corresponding element as the trigger for a DOM event and generates a new seed. This seed includes the URL of the current page and the CSS selector of the element that triggered the event.

The PM does not combine different events to construct new seeds. If there are two events on the web page: Event 1 and Event 2, the crawler will only generate two new seeds $< URL, Event1 >$ and $< URL, Event2 >$ but not $< URL, Event1, Event2 >$ since it cannot track the dependencies among these events. It does not try combinations of events and instead, leaves that task to the ESM.

In the form crawling stage, the PM first collects all forms by identifying the elements with the `form` tag and then tries to submit all of them immediately. For each form, it tries to interact with all the elements inside it sequentially. During the submission, if the PM detects any mutations on the DOM of the form, it will dynamically capture and interact with the new elements. For example, after clicking on the submit button, if a confirmation window pops up, the PM can detect the new elements inside the window and also interact with them.

## B. Evolutionary Search

The design of the Evolutionary Search Module (ESM) in EvoCrawl is inspired by genetic algorithms commonly used in optimization problems. However, traditional genetic algorithms typically aim to find a single optimal solution by iteratively improving the fitness of a population of candidate solutions through natural selection. In contrast, the objective of the evolutionary search module in EvoCrawl is not to seek a single optimal solution but to explore and uncover as many relevant solutions as possible. It searches for sequences that aim to achieve two goals: **explore server-side states and reveal unseen links** within the application.

For each page, the ESM evolves sequences through multiple generations. Each generation includes two steps: "Sequence Generation" and "Sequence Evaluation", where sequences in each generation are generated from sequences that received a high fitness score in previous generations. Therefore, to find sequences that realize the previously mentioned two goals, the genetic algorithm first needs to identify sequences that can lead to optimal descendants. Based on this, we design the fitness function to assign scores to each sequence by using feedback from the browser and web application database during the execution of the sequence. The score reflects the capability of a sequence to generate a new sequence that can satisfy the two goals.

*1) Sequence Generation & Evaluation:* Each sequence consists of multiple genes arranged in a specific order. We define each gene as a web element interaction pair. For instance, when starting with a seed (page URL), the Evolutionary Search Module (ESM) first navigates to the page URL, extracts all interactable elements including elements that listen to JavaScript events, elements belonging to HTML forms, etc. from the page, and constructs genes based on these elements. For example, an "input" HTML element can lead to genes such as "input-click" or "input-typeText"

To generate new sequences, the ESM either randomly combines these genes or performs crossover on previous sequences. For crossover, it selects sequences that received the highest scores from the fitness function in the previous generation and recombines them to create new sequences. For example, if Sequence 1 and Sequence 2 are the sequences with the highest scores, the ESM will concatenate the first half of Sequence 1 with the last half of Sequence 2 to produce a new sequence. Then, the ESM places the submit buttons at the end of each sequence to increase the chances of successfully submitting the filled inputs.

We evaluate a sequence by executing it. The ESM navigates to the page URL and then iterates through all the genes in a sequence. In each iteration, it interacts with the corresponding element based on the interaction type. For example, to evaluate the sequence in Figure 3, it first types texts into "input2" and "input1", clicks on "button1", "a2", and "a1", and finally types texts to the "textarea". After each interaction, the ESM checks the browser's URL field to see if it has been navigated to another page. If clicking on an element navigates the ESM to another page, the ESM will automatically record the new URL, send it to the Page Collection module, navigate back to the previous URL, and continue executing the next gene. By doing this, ESM constrains the search space of the evolutionary algorithm to the current page and thus explores more states of it.

To generate input values for "input" and "textarea" elements, the ESM initially checks for *value* and *placeholder* attributes. The *value* attribute typically stores the default value, and the *placeholder* attribute typically contains a hint of the expected value. If these two attributes do not exist, the ESM heuristically searches for the keywords: URL or email across all attribute values within the elements. If the ESM finds a match, it generates texts conforming to the corresponding format. For example, if the ESM identifies the keyword "URL" in one of the attribute values, it will generate texts: *www.esm*$\{i\}$*.com*, where $i$ is a unique integer used to identify the injected input. Otherwise, it submits a default input that is configured by the user. The default inputs in our experiments are: *esm*$\{i\}$. The ESM's fitness function will then implicitly determine if the generated input meets the element's input constraints.

*2) Dependency Tracking & Enforcement:* During the "Sequence Evaluation" step, if the execution of a gene triggers a JavaScript event and reveals new elements on the page, the ESM infers that the new element depends on the triggered JavaScript event, and can track dependency information by linking the newly revealed elements directly to the gene triggering the changes. We note that due to the non-determinism of pages, some false dependencies may be inferred by the ESM using this heuristic, but in practice, we find that such false dependencies are very rare. In the "Sequence Generation" step, the ESM enforces these tracked dependencies in its mutation function. For example, suppose clicking "button1" triggers a JavaScript event and partially updates the web page, causing anchor element "a3" to appear. Figure 3 represents this sequence update visually. Since "a3" is dependent on "button1", it has been added after "button1" in the example sequence. If multiple elements appear after clicking "button1", the mutation function will randomly select some new elements and add them after "button1". By enforcing the dependency information, the ESM prevents the generation of sequences that violate the order of dependent elements, thereby reducing the search space.

*3) Gene Elimination:* We mentioned that the ESM navigates back to the previous URL if executing a gene causes the ESM to jump to another page while evaluating the sequence. However, navigating to a new page can still disrupt the sequence's execution, as any JavaScript events triggered by the sequence up to that point may be reset due to the page refresh. Consequently, whenever ESM encounters a gene that results in navigating to another page, it will remove this gene and all sequences that contain that gene from the search space. Once removed, a gene is excluded from all subsequent sequences.

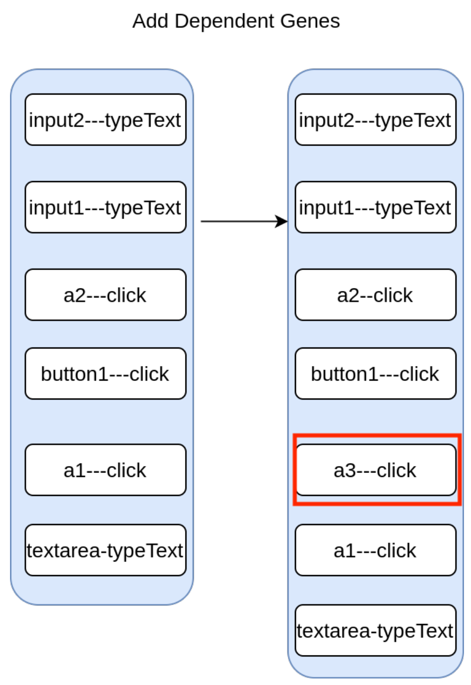


Fig. 3: Mutating Sequences

*4) Fitness Function:* During the "Sequence Evaluation" step, ESM uses a fitness function to find sequences having higher possibilities to generate good sequences in the next generation. Only sequences with the highest fitness scores can survive to the next generation and be used by the ESM to generate new sequences. A good sequence is defined as one that either results in HTML form submissions or the discovery of new links. Specifically, the fitness function leverages feedback from both the browser and the server-side database and employs heuristics to infer which sequence can generate good sequences in the next generation.

The fitness function assigns each sequence an initial, uniform fitness score that undergoes dynamic updates throughout sequence execution. This score represents the possibility of the sequence producing good descendants in the next generation. Actions that could lead to HTML form submissions or induce new elements in the current DOM are rewarded, while actions hindering these objectives are penalized. Notably, actions such as text input to a field, sample file uploads, form submissions, or triggering JavaScript events increase the fitness score. A successful form submission is determined by inspecting the server-side web application's database after executing the entire sequence. To detect this, the input text generated by the ESM for each input field contains a unique tainted value. ESM then queries whether the tainted value is injected into the database by the executed sequence. This injected text serves as an indicator of a successful form submission. Conversely, if the ESM executes a gene but the corresponding element is currently invisible, the fitness function will punish the sequence, as this indicates an incorrect order of the elements' interactions. The fitness function is computed as:

TABLE I: Fitness Function Weights and Objectives

| Weight | Value | Objective ($o_i$) |
|---|---|---|
| $w_1$ | 40 | Number of Form Submissions |
| $w_2$ | 20 | Number of Filled Inputs |
| $w_3$ | 20 | Number of Uploaded Files |
| $w_4$ | 15 | Number of Triggered JS Events |
| $w_5$ | -2 | Number of Invisible Elements |

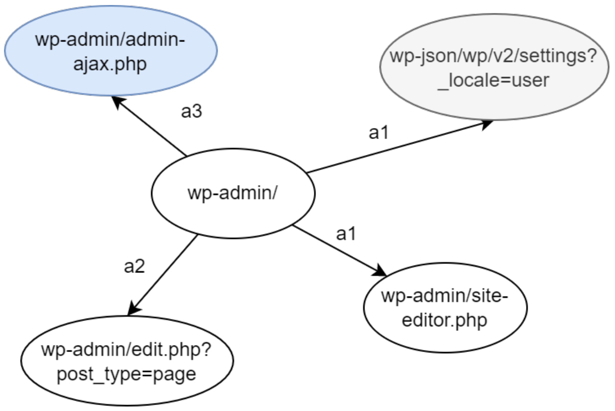
$$f = \sum o_i \cdot w_i \tag{1}$$

$o_i$ represents the objective variable, while $w_i$ denotes the associated weight parameters. The weight values and their corresponding objectives are presented in Table I. We found empirically that this set of weights successfully helps the EvoCrawl outperform other crawlers, so we did not fine-tune them for each application. However, these weights are tunable and can be further studied in future work.

We design the fitness function to reward sequences whose genes trigger JavaScript events during execution because these JavaScript events can dynamically update the webpage and reveal previously unseen elements, such as those in forms or containing new links. We want such sequences to survive to the next generation, enabling the ESM to explore the new elements introduced by the JavaScript events these sequences trigger.

The fitness function significantly rewards sequences that successfully submit a form. This incentivizes the function to find sequences that not only fill in essential fields but also include optional ones, thereby inserting more data into the database. For instance, consider a short sequence discovered by the ESM: "filling in field A and clicking the submit button." If this sequence leads to a successful form submission, it receives a high score due to the substantial reward for "form submission." In subsequent generations, many sequences are generated based on this initial sequence. One resulting possibility could be: "filling field A, filling field B, and clicking the submit button" (assuming A and B belong to the same form). If this new sequence also successfully submits the form, it earns a higher score than the previous one because of the additional reward for "typing text into more input fields." Consequently, in the next generation, newer sequences are generated based on this enhanced sequence. Through this iterative process, EvoCrawl can find sequences that fill in more input fields for each form, resulting in a diversity of server-side states.

The fitness function also helps EvoCrawl avoid sequences that include fields with constraints it cannot satisfy. Suppose there is a field C whose input constraints EvoCrawl is unable to satisfy. Any sequence that includes field C will receive a lower score than the short sequence. This is because the amount it loses for failing to attain the "form submission" reward outweighs the amount it gains for "typing text into inputs." Consequently, EvoCrawl will prioritize sequences that do not include field C.



wp-admin/admin-ajax.php
wp-json/wp/v2/settings?_locale=user
a3
a1
wp-admin/
a1
a2
wp-admin/site-editor.php
wp-admin/edit.php?post_type=page

Fig. 4: An Example Sitemap for WordPress - Blank Node: Page URL, Grey Node: Restful API, Blue Node: Ajax URL

## C. Vulnerability Detectors

EvoCrawl is designed to allow modular integration of vulnerability detectors, which are utilized as EvoCrawl explores the application. In this study, we demonstrate EvoCrawl's ability to detect authorization and injection vulnerabilities, which have become increasingly prevalent [10]. In particular, we search for Insecure Direct Object References (IDORs) and Cross-Site Scripting (XSS). These two detectors operate independently and can be executed either individually or in conjunction with each other.

*1) XSS Vulnerability Detector:* We integrate the XSS Vulnerability Detector (XVD) directly into both the ESM and the PM. For the PM, after each submission of a form, the XVD replaces the input value with its XSS payload. For the ESM, the XVD directly replaces all text generated by the evolutionary crawler with the XSS payload.

The XSS payload of our XVD is similar to the payload used by BlackWidow. If the payload injected into each input field is successfully executed by the JavaScript Engine of the browser, a unique integer will be pushed into a global list that is pre-inserted into the HTML header of the web page. The unique integer is generated using a UNIX timestamp to avoid two input fields being inserted with the same value. By checking the values in the global list, we can know which payload has been executed and further trace it to the source input field.

*2) IDOR Vulnerability Detector:* Unlike XVD, we couldn't find an existing IDOR vulnerability detection tool that we could easily integrate with EvoCrawl, so we designed our own. The IVD utilizes two users with distinct access control levels: an admin user and a non-admin user. We assume that any page or resource accessible via UI navigation should be accessible to the current user. The IVD initiates the process by employing the admin user to crawl and collect resources within the web application. Subsequently, it utilizes the non-admin user to identify and exclude public resources accessible via the UI for both user types. Finally, the IVD assesses the access control level of private resources (those not accessible to the non-admin user via the UI). It does so by attempting to access these private resources directly using the non-admin user's credentials. If the non-admin user can access these private resources, despite their exclusivity to the admin user's UI, it signals potential vulnerabilities within the system.

We first describe how the IVD collects resources and filters out public ones. During the crawling process of PM and ESM, the IVD automatically captures all requests sent from the browser, extracts the request URLs, and builds them into a sitemap. EvoCrawl's sitemap thus captures all navigation paths between URLs found during crawling, including those that result from interacting with JavaScript events, anchor elements, and any other HTML elements that EvoCrawl interacts with. Each node in the sitemap represents a URL and the edge between the two nodes represents the HTML element that triggers the transition from the source URL to the destination URL. Figure 4 is an example sitemap for WordPress.

Then, the IVD uses another user with a different privilege level from the crawler user to test if each edge in the sitemap is accessible and also marks the corresponding edge by replaying every interaction of both the crawler module and evolutionary search module. If the replayer (a module inside the IVD) manages to replay the interaction by using the related element's CSS selector, the corresponding edge will be labeled as accessible (i.e. public) to the replayer and vice versa. It is important for the replayer to correctly replay the interaction and mark the right edge since mistakenly labeling an edge can cause the IVD to misclassify an object, which leads to both false positives and false negatives. We also note that it is important that both the crawler and replayer operate on the same web application instance, in lockstep. This is because we need the replayer to see the same web application state as the crawler, to avoid missing public resources. For example, the crawler could create a public object and then subsequently delete it. If the replayer tries to access the same object on another instance, or outside of lockstep with the crawler, it might mistakenly believe the object to be private, and this will lead to a false positive when it finds it is able to access it later during detection.

After the replayer tests all the edges, the IVD can know which resources are private by parsing the sitemap. Considering the example in Figure 4, if element "a2" can be accessed by the replayer while element "a1" cannot, the `edit.php?post_type=page` is considered a public resource while the `site-editor.php` is considered private. There might be different paths to reach the same resource. If one of the paths can be accessed by the replayer, the destination node will be considered public. After gathering all the private URLs, the IVD will directly send forged requests to them and check their access control levels by analyzing received responses.

After collecting private resources, it tests whether they can be accessed by an unprivileged user. To do this, IVD sends three forged requests to each private resource as three different users. Then, the responses are parsed to decide whether each resource is appropriately protected. We refer to this as the triad test. In the triad test, the session cookies are obtained after automatically logging in with the user's credentials. The first, userA, is the user used by the crawlers. The second, userB, is the user that was used during replaying, and the third, userC, must be at the same privilege level as userB. The IVD uses two steps to determine whether the responses disclose private information to the attackers. The first step uses keyword matching while the second step compares responses. Empirically, for each application, we observed that most "access denied" responses share common sentences. We use these sentences to identify responses with proper access control. Based on our experiments, 5 sentences are enough for each tested application. We call these sentences access-denied sentences. If the responses received by the attackers do not contain any access-denied sentences, they are further passed to the second step parser to decide whether there are broken access controls. For now, we manually collected the access-denied sentences for each application. It is possible that we failed to capture all the denied sentences. In this case, we rely on the second step parser to check the access controls.

For the second step parser, simply comparing userA's response and userB's response is insufficient, because we would not know if the response-differences comes from the page contents or just user-specific data such as username, and user email on the webpage. This is where userC is needed. Since both userB and userC are at the same privilege level and neither must have access to the private resources, their responses must be similar with differing in only user-specific content. Thus, we can detect the user-specific data and ignore it when comparing userB's response with the userA's response.

## IV. Implementation

EvoCrawl is built using a customized version of TestCafe [11]. TestCafe is an end-to-end web application testing framework. It provides browser automation capabilities along with useful features like capturing all requests-responses sent and received, checking if an HTML element is visible on screen, interacting with an element, and running the JavaScript code in the browser.

We inject the rrweb script—a tool designed for recording and replaying user interactions on the web—into the header of each page. This integration enables the capture of newly visible elements for the PM and the ESM, as well as the recording of interactions for IVD. The rrweb's recording module can capture any mutations happening on the current page with low overheads by using the MutationObserver function.

EvoCrawl uses Kafka as a durable queue for storing seeds. Both modules publish newly discovered page URLs to the queue and consume from it using separate consumer groups, simulating multiple queues.

ESM and PM operate as separate processes with distinct cookie sessions. Applications like Opencart or phpBB include tokens in their page URLs that need to be matched with the corresponding cookie values. When exchanging seeds (page URLs) between ESM and PM, the token values must be automatically replaced with the appropriate ones for each module. To achieve this, both modules perform two tasks:

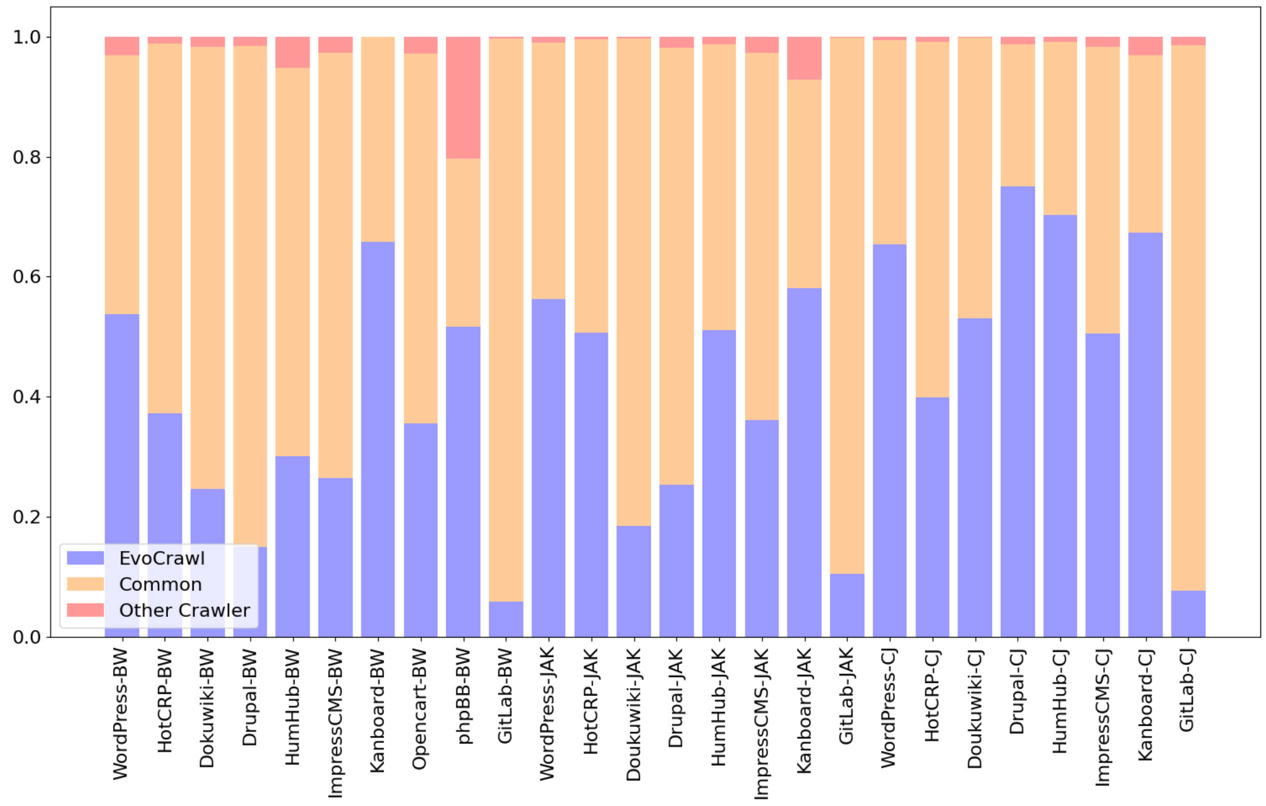


Fig. 5: Each bar presents the results of EvoCrawl compared with another crawler. The blue bar represents the unique lines covered by EvoCrawl, the orange bar denotes the common lines covered by both crawlers, and the red bar shows the unique lines covered by another crawler. (BW-BlackWidow, JAK-JAK, CJ-CrawlJAX)

identifying token names and replacing their values. We compare the redirect URLs after logging in from both modules to determine token names. By comparing the query strings of the URLs, we infer which parameters differ and consider them as tokens. Once token names are obtained, the modules extract their values from the URLs and replace tokens in incoming seeds. However, this approach only works for persistent tokens present in all page URLs. Tokens that appear in only some URLs cannot be identified and captured by EvoCrawl, leading to the exchange of invalid seeds. While this slows down EvoCrawl, it does not break the entire system.

### A. *rrweb*

rrweb consists of two modules: recording and replaying. The recording module assigns unique rrweb-IDs with timestamps to DOM elements for event tracking. The replaying module replays interactions based on the recorded rrweb-IDs and timestamps. However, when EvoCrawl's public filter module replays interactions as another user, the rrweb-IDs may lead to incorrect elements due to dynamic DOM changes. To address this, we only use rrweb's recording module and implement our replaying mechanism ourselves.

### B. *Initial Setup*

We need to configure the application to have basic users and enable automatic logins to the application. This minor, one-time manual effort to register three users is a prerequisite for running EvoCrawl on a web application.

## V. Evaluation

In this section, we present an empirical evaluation of EvoCrawl. We evaluate EvoCrawl along two metrics: the amount of code coverage attained and the number of HTML forms successfully submitted. The latter is an indicator of the number of server-side web application states EvoCrawl is able to explore. We evaluate the code coverage of EvoCrawl against three state-of-the-art academic crawlers: BlackWidow, JAK, and CrawlJAX. We exclude the scanners in the web security community such as skipfish [12], Arachni [13], and w3af [14] etc., since BlackWidow [6] has previously demonstrated significant improvement over them. We assessed all 4 crawlers on 10 modern web applications. We evaluate the ability of EvoCrawl to submit HTML forms against BlackWidow since it is the best-performing crawler and the only one specifically designed to explore the server-side state.

TABLE II: Applications for Experiments

| | Functionality | Version | Github Stars | Used in other work |
|---|---|---|---|---|
| WordPress | Blog | 6.4.3 | 2.3k | [6], [7] |
| HotCRP | Content Management System | v3.0b3 | 319 | [6] |
| Dokuwiki | Content Management System | 2022-07-31"Igor" | 4k | |
| Drupal | Content Management System | 9.3.15 | 4k | [6] |
| Humhub | Social Software Platform | 1.12.1 | 6.2k | |
| Opencart | eCommerce | 4.0.0 | 7.3k | |
| phpBB | Forum | 3.8.8 | 1.8k | [6], [8], [7] |
| ImpressCMS | Content Management System | 1.4.4 | 27 | |
| Kanboard | Project Management System | 1.2.22 | 8.2k | |
| Gitlab | DevSecOps Platform | 11.5.1 | 23.6k | |

### A. Experiment Setup

Each crawler runs on a 4-CPU virtual machine with 6GB memory. The CPU type of the virtual machine is Intel(R) Xeon(R) Gold 6336Y. To guarantee a fair comparison, we reset all tested web application instances before each crawling session. This ensures that all scanners commence from the same initial state, minimizing any potential disparities caused by differing application states.

For coverage experiments, since EvoCrawl uses both the page collection module and the evolutionary search module to interact with a web application, we also run two processes in parallel for other crawlers to let them have the same CPU resources as EvoCrawl. We used lines of code as a metric for coverage and generated a coverage report indicating which lines had been hit for each request sent to the server. The coverage report for the PHP application is generated by using Xdebug [15] and php-code-coverage [16]. For the application in Rails production, we use Coverband [17] to collect coverage results. Also, we disabled the vulnerability detectors for all scanners since we want to focus on testing the ability to crawl the web application.

To detect a successful form submission, we record the text that each crawler filled into each form and log all transactions that modify the database tables. If the text filled by the crawlers appears in any of the transactions, we consider the corresponding form to be successfully submitted.

The public filter of IVD requires an attacker to replay the interactions of both the crawler module and the ESM to classify the collected URLs. We set the privilege level of the attacker to be the second highest and the crawler user to be the highest.

As for the configurations for the tested crawlers, we manually set up the login credentials for all of them and prevented them from crawling on the user page, the basic configuration page, and the extension/plugin installation page of each web application, because crawling on those pages may change the login credentials or cause the web application to crash. Moreover, we prevent all crawlers from interacting with logout buttons to make sure they always stay logged in. Each testing process was run for 24 hours. In addition, we ran EvoCrawl 5 times because of the randomness of the evolutionary algorithm.

For the ESM, all the parameters including the sequence length, the number of generations, etc. are fixed for all the benchmarks. We use the default settings for CrawlJAX with unlimited crawling depth and states. We also enable it to click on event handlers. For JAK, we follow the same configuration the developers provided in the example file. For the form submission experiment, since we need to search the injected text to detect whether the form has been successfully submitted, we need scanners to generate unique text for each field. While EvoCrawl already supports this, BlackWidow generates the same texts for all the fields. Hence, we modify BlackWidow's implementation to support unique text injection. This modification is exclusive to the form submission experiment and does not apply to the coverage experiment, where we use BlackWidow's original design.

In selecting targets on which to evaluate EvoCrawl, we sought applications that were both representative and had been used in other academic research. For this evaluation, we define a representative set of applications as 1) representing a variety of functionalities and 2) having an active user base and being actively maintained. Table II provides information on our set of selected applications. The "type" column in the table indicates the diversity of the applications, while the number of GitHub stars approximates their user base. Furthermore, all applications are in their latest versions and are actively maintained at the time of writing. Finally, we cite other works that have also used the particular application in other studies.

### B. Code Coverage

Figure 5 presents the final code coverage achieved by each crawler across all tested applications after a 24-hour run which is presented proportionally. We provide the absolute values for the coverage results in the Appendix. The blue bar represents the number of unique lines covered by EvoCrawl, the orange bar indicates the number of lines covered by both crawlers, and the red bar shows the number of unique lines covered by other crawlers. We do not include results for JAK and CrawlJAX on Opencart and phpBB, as these crawlers cannot handle the token implementations of these web applications. EvoCrawl has the highest coverage on all the tested applications over other scanners. Even for the next best scanner BlackWidow, EvoCrawl outperforms it and has an improvement ranging from 6% to 192% across different

TABLE III: This Table presents p-value results for each application between EvoCrawl and BlackWidow, with both EvoCrawl and BlackWidow run 5 times on each application.

| | WordPress | HotCRP | Dokuwiki | Drupal | Humhub | ImpressCMS | Kanboard | Opencart | phpBB | GitLab |
|---|---|---|---|---|---|---|---|---|---|---|
| p-value | 0.00096 | 0.000007 | 0.00082 | 0.004448 | 0.000031 | 0.000502 | 0.000719 | 0.000009 | 0.000159 | 0.000120 |

applications. Table III presents p-values comparing EvoCrawl with BlackWidow, highlighting significant differences between them.

*1) Case Studies of the Coverage Results:* We include a case study of why EvoCrawl achieves better coverage than BlackWidow.

**HotCRP**. BlackWidow achieves lower coverage on HotCRP because it hits the "cancel" button before reaching the "save" button during the submission of certain forms, while the ESM of EvoCrawl can find the sequence of interactions that omit the cancel button but click on the "save" one. This is especially important for HotCRP, as the crawler must submit certain forms before exploring related code blocks. For example, a crawler needs to first successfully submit a paper, before it can successfully crawl on the "reviews for the paper" page. Moreover, BlackWidow fails to enter the correct values for some input fields inside certain forms. For these forms, the ESM can find the sequence that leaves these difficult input fields blank and only fills in fields that heuristics can infer.

**Kanboard**. EvoCrawl is the only scanner that successfully creates tasks inside the projects on Kanboard. Similar to HotCRP, there are input fields inside the task creation form whose values cannot be resolved by all the scanners. Instead of filling in the wrong values like other scanners do, the ESM of EvoCrawl successfully finds sequences that leave these fields blank and manages to create the tasks, thereby further executing the code related to task modification and management.

**WordPress**. EvoCrawl's improved coverage on WordPress mainly comes from two factors. First, the evolutionary search module of EvoCrawl manages to install different themes on WordPress and further explores the code blocks of these themes. Second, although BlackWidow can create draft posts on WordPress, it fails to publish them, since publishing posts requires the crawler to trigger a JavaScript event after filling in the form. EvoCrawl is the only crawler that finds this sequence of interactions, while BlackWidow fails to find it.

**Opencart**. One of the factors contributing to BlackWidow's lower coverage is its inability to submit forms in Opencart. For successful form submission, the crawler must trigger a JavaScript event after filling in the input fields. However, BlackWidow's strategy of enumerating all possible combinations of JavaScript events and HTML forms results in an excessively large search space. This vastness prevents it from identifying the correct sequence needed to submit the form. Additionally, certain page links within Opencart remain hidden until the crawler triggers a combination of JavaScript events, which BlackWidow fails to find.

TABLE IV: This Table presents the results of the HTML form submission. The data includes Unique forms submitted by EvoCrawl, Common forms submitted by both crawlers and Unique forms submitted by the BlackWidow

| | Unique-EvoCrawl | Common | Unique-BlackWidow |
|---|---|---|---|
| WordPress | 8 | 7 | 2 |
| HotCRP | 17 | 6 | 3 |
| Humhub | 25 | 3 | 2 |
| Drupal | 70 | 11 | 33 |
| Kanboard | 17 | 5 | 0 |
| phpBB | 15 | 12 | 10 |
| ImpressCMS | 7 | 2 | 3 |
| Opencart | 15 | 0 | 1 |
| Dokuwiki | 6 | 8 | 2 |
| Gitlab | 30 | 1 | 1 |

For other applications, EvoCrawl generally has better results for two reasons. First, it does not spend time trying combinations of unrelated events. Therefore, it has time to extensively navigate the application and interact with more pages and forms than BlackWidow does. Second, for certain forms, EvoCrawl is the only tool that finds the correct sequences to submit inputs. The two factors together lead to a higher coverage achieved by EvoCrawl in terms of the overall exploration of the application.

### *C. HTML form submissions*

Table IV presents the number of successfully submitted HTML forms, which is an indication of how well EvoCrawl explores server-side state. We only compare EvoCrawl with BlackWidow, since the other two crawlers are not specifically designed to explore the server-side states.

We monitor all the data transactions happening on the server side and track all the inputs inserted into the database. We further collect which HTML forms have been successfully submitted during scanning while using the `action` attribute to represent each form. We do not collect submissions other than HTML forms because they are difficult to track.

For all applications, EvoCrawl is able to submit more forms than BlackWidow, one of the reasons that EvoCrawl submits more forms than BlackWidow is due to EvoCrawl's efficiency. The genetic algorithm and dependency tracking enable EvoCrawl to search intelligently and spend less time on each page. Consequently, it has more time to explore additional pages, thereby discovering and submitting more forms.

Upon closer inspection, we find that EvoCrawl outperforms BlackWidow on HotCRP, Humhub, and Kanboard because it can bypass optional fields with strict constraints. BlackWidow

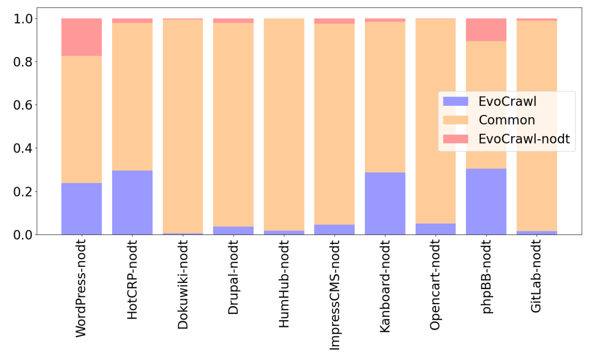


Fig. 6: Each bar presents the results of EvoCrawl compared with EvoCrawl without dependency tracking. The blue bar represents the unique lines covered by EvoCrawl, the orange bar denotes the common lines covered by both crawlers, and the red bar shows the unique lines covered by EvoCrawl with no dependency tracking.

fails to submit these forms because it either enters the wrong values to certain input fields or interacts with elements in the wrong order. Moreover, it further misses the forms that depend on the successful submission of these previous forms. For example, BlackWidow fails to submit a paper on Hotcrp and therefore cannot detect the forms that assign the paper and review the paper. It also fails to create tasks inside the project page on Kanboard, and cannot create new spaces on Humhub as well.

There are cases where BlackWidow submits forms that EvoCrawl does not in our experiments (i.e. in Drupal and phpBB). We found that this is a result of differences in seed scheduling between BlackWidow and EvoCrawl, which causes them to explore slightly different portions of the applications, and thus interact with different forms. The forms that BlackWidow submitted were not analyzed by EvoCrawl. Overall, EvoCrawl still submits more forms than BlackWidow, indicating that EvoCrawl has a faster crawling speed and form submission rate than BlackWidow.

### D. *Benefits of Dependency Tracking*

As discussed in Section III, the ESM employs dependency tracking to reduce the search space. To evaluate the effectiveness of this feature, we implement EvoCrawl-nodt, which is a version of EvoCrawl with dependency tracking disabled, so that no dependency information is used during sequence generation. Figure 6 illustrates the code coverage of EvoCrawl-nodt compared with EvoCrawl.

For most applications, dependency tracking largely enhances code coverage. However, for some applications, the coverage results between the two configurations are similar. Upon further inspection, we discovered that these applications do not heavily rely on JavaScript events to reveal links or forms, unlike others. Consequently, dependency tracking does not substantially increase coverage for these applications.

TABLE V: Results of Vulnerability Detection Experiments

| | WordPress | Humhub | ImpressCMS | Kanboard |
|---|---|---|---|---|
| EvoCrawl | 2/2 | 0/1 | 2/2 | 2/3 |
| BlackWidow | 1/2 | 0/1 | 0/2 | 1/3 |

For phpBB and WordPress, EvoCrawl-nodt executes a notable amount of lines that EvoCrawl does not cover. This discrepancy is mainly because EvoCrawl and EvoCrawl-nodt crawl on different sets of pages. The dependency tracking allows EvoCrawl to find sequences that respect the order of the web elements. This enables EvoCrawl to find new elements that only the execution of these sequences can reveal. If these new elements include anchor elements with links to new pages, EvoCrawl will add them to the queue and crawl them, while EvoCrawl-nodt does not find these and ends up crawling other pages. Consequently, this results in longer queues for EvoCrawl than EvoCrawl-nodt, as some pages are only in the queue of EvoCrawl but not in the queue of EvoCrawl-nodt. Due to the large number of web pages in phpBB and WordPress, EvoCrawl cannot finish crawling all the pages within the 24-hour limit, thereby failing to crawl on the pages at the end of the queue. However, the total coverage of EvoCrawl is always larger than that of EvoCrawl-nodt.

## VI. Vulnerabilities

We now evaluate the ability of EvoCrawl to detect vulnerabilities, as well as detail the new vulnerabilities that EvoCrawl has discovered.

### A. *Experiments on Known XSS Vulnerabilities*

To compare EvoCrawl's detection ability on known XSS vulnerability detection with other crawlers, we conducted 24-hour experiments on both EvoCrawl and BlackWidow using vulnerable versions of web applications. To do this, we selected versions of web applications with previously found vulnerabilities that were documented in enough detail that we could 1) reproduce the environment and conditions under which the vulnerabilities can be triggered; 2) confirm that triggering the vulnerabilities does not require a crafted payload that bypasses sanitizers, since both EvoCrawl and BlackWidow specialize in finding injection points, but not in crafting payloads to bypass sanitizers; and 3) have not been previously found by either EvoCrawl or BlackWidow, ensuring that both tools have an equal chance of detecting the selected vulnerabilities. We used the CVE Details website [18] to find information on known vulnerabilities. The specific versions tested were WordPress-4.7.2, Kanboard-1.2.8, ImpressCMS-1.4.4, and Humhub-1.11.0.

Table V presents the results of the known vulnerability detection experiments. EvoCrawl outperforms BlackWidow on all tested applications. Below, we provide a detailed analysis of the vulnerability detection performance of EvoCrawl and BlackWidow for each application.

**Humhub.** Both EvoCrawl and BlackWidow fail to detect the vulnerability because it requires the crawlers to re-login as another user to manifest. Although both crawlers successfully injected payloads into the injection point, which is the "name" field of the Humhub Space, they could not complete the necessary steps to detect the vulnerability.

**Kanboard.** EvoCrawl successfully detects 2 out of 3 vulnerabilities in Kanboard. The vulnerability that both EvoCrawl and BlackWidow fail to detect, similar to the one in Humhub, requires the crawler to re-login as another user. The vulnerability detected only by EvoCrawl requires the crawler to first create a "task" under a "project" in Kanboard and then inject the payload into the "external link" field within the task. As mentioned in subsection V-B, BlackWidow fails to leave some fields blank when submitting the form to create tasks. Consequently, it cannot submit the task creation form and detect this vulnerability.

**ImpressCMS.** EvoCrawl successfully captures two vulnerabilities in ImpressCMS on the "edit user" page and the "blocks admin" page. For the first vulnerability, BlackWidow fails to bypass the input constraints of a field within the form, while EvoCrawl finds the sequences of interactions that leave it blank. For the second vulnerability, the form containing the injection field exists in a hidden part of the application; BlackWidow fails to find the correct interaction sequences to reveal it.

**WordPress.** Both EvoCrawl and BlackWidow find the vulnerability in the "taxonomy name" field, but BlackWidow fails to find the vulnerability in the "upload filename" field. This is because BlackWidow attempts to iterate through all the combinations of JavaScript events, which slows down the crawler.

## *B. Zero-day XSS Vulnerabilities Detection*

We test EvoCrawl using the latest versions of benchmarks previously used in our coverage experiment our coverage experiment. This demonstrates the ability of EvoCrawl to uncover Zero-day XSS vulnerabilities.

EvoCrawl has identified 5 zero-day XSS vulnerabilities across 10 web applications, all of which have been reported. Among these, a vulnerability in HotCRP and a vulnerability in Kanboard have been patched. Two WordPress vulnerabilities have been acknowledged, yet they won't be addressed as the injection point for the XSS attack is not included in the threat model that WordPress developers consider. The second vulnerability in HotCRP will not be fixed as the requirements for exploitation are outside what the developer considers the expected usage model of HotCRP.

For WordPress, EvoCrawl identifies two stored XSS vulnerabilities, which can only be exploited by admin or editor users. The developers have decided not to address these vulnerabilities as admins and editors are considered trusted in their threat model.

In HotCRP, we find one stored XSS vulnerability and one reflected XSS vulnerability. The stored XSS vulnerability has been acknowledged by the developers and will be fixed in future versions of the application. The reflected XSS bug in HotCRP, however, cannot be exploited by attackers as it is only visible to admin users and protected by a CSRF token.

For Kanboard, EvoCrawl successfully identifies one stored XSS vulnerability, which has been reported to the developers and will be patched in future versions.

EvoCrawl generates one false positive in Humhub. EvoCrawl detects one injection field that allows the website owner to inject a custom script for tracking page statistics. Since the web developer intentionally designed the field to accept *script* as input, we conservatively counted this as a false positive for EvoCrawl.

## *C. IDOR Vulnerabilities*

Table VI presents the results of the IVD evaluation. To assess its performance, we collected a variety of metrics. These include the total number of URLs discovered by EvoCrawl, the number of URLs classified as private, the number of URLs classified as public, and the number of false positives. Additionally, we categorized vulnerable endpoints into two groups: those arising from the site builder's incorrect configuration or privilege settings (Vul-Type1), and those resulting from improper code implementation by web developers (Vul-Type2).

As described in the design section, the IVD relies on a sitemap to classify resources as public or private. Resources reachable from both the admin and unprivileged users' UI are classified as public. Conversely, if the paths exist only in the admin UI, IVD classifies the resources as private. However, some public resources may also lack paths in the unprivileged UI of the application, leading the IVD to misclassify these public resources as private. Because these resources are in fact public, when the IVD later finds that they are accessible to unprivileged users and reports them, the resources will result in false positives. This issue is particularly evident in applications like phpBB and HotCRP, which account for 374 out of 385 and 35 out of 39 false positives, respectively. For the remaining false positives, the reason is that the crawler itself is not able to find all existing navigation paths for a public resource within the allotted time.

Regarding Vul-Type1, the IVD has uncovered multiple endpoints that directly expose application resources such as images and JavaScript files. For example, some of the detected endpoints resemble `http://localhost/path/sample.png`. Attackers could access these resources by sending requests targeting these endpoints without authentication. Although the application code itself does not cause these vulnerable endpoints, we still believe it is important for the crawler to identify such endpoints. This capability helps site builders ensure that the privilege settings for each folder are configured appropriately.

For Vul-type2, the IVD discovers 3 vulnerable endpoints across 10 web applications. All of these vulnerabilities have been reported. In the case of ImpressCMS, one vulnerability has been acknowledged and the patch for it is currently under development. Another vulnerability is still under inspection.

TABLE VI: IDOR Vulnerability Detector Results

| | URLs | Private URLs | Public URLs | FP | Vul-Type1 | Vul-Type2 |
|---|---|---|---|---|---|---|
| WordPress | 1025 | 379 | 646 | 9 | 106 | 0 |
| HotCRP | 526 | 415 | 111 | 39 | 3 | 0 |
| Humhub | 10729 | 9451 | 1278 | 0 | 5 | 0 |
| Drupal | 1908 | 1242 | 666 | 4 | 55 | 0 |
| Kanboard | 7973 | 4511 | 3462 | 17 | 0 | 0 |
| phpBB | 1684 | 1527 | 158 | 385 | 0 | 0 |
| Opencart | 1202 | 870 | 332 | 4 | 60 | 0 |
| Dokuwiki | 3121 | 864 | 2257 | 8 | 13 | 0 |
| ImpressCMS | 615 | 593 | 22 | 0 | 111 | 2 |
| Gitlab | 1382 | 640 | 742 | 27 | 63 | 1 |

The vulnerability in Gitlab has been reported and will be addressed in a future version.

### D. Summary of New Vulnerabilities Found

In total, EvoCrawl has identified eight vulnerabilities in popular web applications such as WordPress, HotCRP, Kanboard, ImpressCMS, and Gitlab. Out of these, six vulnerabilities have been acknowledged and confirmed by the developers. The details of each vulnerability are as follows:

- WordPress (acknowledged but not fixed): The two XSS injection points of WordPress are the comment field and the post title field. Both of these fields lack proper sanitization, allowing editor users or admin users to inject custom scripts into them. According to the WordPress security policy, XSS injection points that can only be exploited by higher-level users will not be fixed.
- HotCRP (acknowledged and fixed) [19]: One XSS injection point on `settings/decisions` page. Chair or admin users can inject custom scripts into the decision name field.
- HotCRP (reported but not acknowledged): EvoCrawl identified a reflected XSS injection point on the `settings/reviews` page, specifically in the round name field. However, this injection point is only accessible to admin users and is protected by a CSRF token, so the developer does not consider it a vulnerability.
- Kanboard (acknowledged and fixed) [20]: One XSS injection point on `settings/api` page, enabling admin users to inject scripts to the application URL field.
- ImpressCMS (acknowledged and being fixed): An IDOR vulnerability on endpoint `userinfo.php?id=1`. Attackers can acquire other users' information by changing the value of the `id` parameter.
- ImpressCMS (reported and still under inspection): An IDOR vulnerability on endpoint `/libraries/image-editor/image-edit.php?image_id=1&uniq=`. Attackers can force browsing to the private images by changing the `image_id`.
- GitLab (acknowledged and fixed): An IDOR vulnerability on endpoint `autocomplete/users.json?search=&active=true&current_user=true`. The AJAX request targeting at this endpoint reveals all users' information including avatar URL, username and states, etc.

## VII. Limitations

**Parameter Tuning.** For optimal results, EvoCrawl currently requires manual parameter adjustments within the fitness function. In the future, we aim to conduct a more comprehensive analysis of the influence of each parameter. Our goal is to design a system that can autonomously fine-tune these parameters, enabling adaptation to the specific needs of each tested application.

**Seed Selection.** In many cases, various URLs can direct to the same or highly similar pages within applications. For example, pages displaying objects with different sorting criteria may possess distinct URLs. As EvoCrawl relies on page URLs as seeds for crawling, there's a potential for redundancy, where the tool might spend time crawling pages it has already processed, thereby decreasing efficiency further. In future developments, we will try to implement more effective methods for distinguishing between different pages. Instead of relying solely on page URLs, we aim to employ techniques such as DOM comparison to achieve greater accuracy and precision.

## VIII. Related Work

### A. Access Control Vulnerability Scanners

[4], [5], [3], [21] detect access-control vulnerabilities in a white-box manner. However, only doing state analysis can lead to missing certain links, since some of them are generated during the run time. Overall, white-box methods are always limited by the language of the source code, therefore, making it hard to generalize for all websites or web applications.

Yelp's Fuzz-lightyear is a framework designed to automate IDOR discovery through stateful fuzzing [22]. It leverages the Swagger or OpenAPI specifications of a web application, first proposed in the RESTler paper by Atlidakis et al. [23]. RESTler was designed to be a generic bug-detecting tool. Therefore, it can only detect the bugs that cause the app server to respond with an HTTP 500 (Internal Server Error) code and cannot detect if a purposely formed malicious request succeeded.

AuthScope by Zuo et al. [24] has a similar approach to EvoCrawl's IVD in that it focuses on automatically executing a mobile app and detecting vulnerable authorizations. They perform differential traffic analysis to recognize the protocol fields in the request structure which are then automatically substituted to check for correct authorizations. They also develop a targeted dynamic activity explorer to automatically log in to the app and explore the app activities in a prioritized depth-first search approach to get post-authentication messages. This works well for mobile apps due to the layered structure of the in-app activities. They, however, do not handle dynamic resource generation. Furthermore, they assume that all post-login resources are private and therefore may still have a lot of false positives even after pruning public activities/interfaces accessible prior to login.

### B. Web Crawlers

BlackWidow by Eriksson et al. [6] and jAk by Pellegrino et al. [8] also build their navigation graph with client-side JavaScript events and HTML forms. However, they search the target sequences by enumerating the nodes in their navigation graph, causing the crawlers to waste extra time exploring events that are not related, decreasing the overall performance. Furthermore, they capture the JS events by hacking the `addEventListener` function dynamically each time a page has been loaded. This method is not robust and sometimes misses events on the page.

Enemy of the State by Doupé et al. [7] uses static links and forms to build the navigation graph but misses the JavaScript events on the client side. Since many modern web applications heavily rely on SPA (single-page applications) and AJAX techniques, it is hard for Enemy of the State to fully explore these websites.

Crawljax by Mesba et al. [9] uses a state machine to guide the crawling process. Each state represents a unique DOM (Document Object Model) of the web page. However, CrawlJAX cannot track dependencies among different states.

There are other black-box scanners [25], [26] but they mainly focus on vulnerability detection. Deemon [25] is able to detect the CSRF vulnerability by modeling the behavior of the application, while Pellegrino and Balzarotti [26] proposed an automatic tool to detect logic errors with analysis on interaction traces.

Recently, grey-box fuzzing techniques have gained traction for testing web applications [27], [28], [29]. However, it is worth noting that [27], [28] are specifically designed for testing PHP applications, limiting their applicability. Additionally, the effectiveness of Witcher [29] is related to the performance of the black-box scanner it incorporates.

### C. Evolutionary Search in Web

Attwood et al. [30] summarize some works that use evolutionary search in web security, but none of them use it to crawl web applications. [31] and [32] both use evolutionary algorithms to generate payloads that are more likely to pass through the server sanitizer and find an XSS vulnerability. However, the evolutionary algorithms are used to generate input payloads and not for crawling. On the other hand, EvoCrawl's goal is to try revealing as many sources as possible but not generate inputs that successfully pass sanitizer checks.

### D. Machine Learning in Web Scanning

Lee et al. [33] propose Link, a black-box scanner that applies reinforcement learning to adapt the generated XSS payloads for each input field by observing the received responses. Link iterates through URL-Parameter pairs and adapts payloads based on the received response after each attack. However, it mainly focuses on adapting the payloads while EvoCrawl tries to maximize the code coverage of an application.

Mind2Web [34] is a generalist web agent that utilizes Large Language Models (LLMs) to complete tasks based on language instructions and by parsing HTML. However, it still requires specific language instructions for each task, which limits its ability to automatically scan applications. Consequently, LLM-based web agents have not yet been a good fit for automatic vulnerability detection.

## IX. Conclusion

Our experiments show that using an evolutionary search algorithm in conjunction with dependency tracking enables EvoCrawl to perform a fine-grain search of web applications. This enables EvoCrawl to attain greater code coverage and submit more inputs to web applications than previous approaches. In particular, we use evolutionary search to generate sequences of web interactions to target certain favorable events, such as successfully submitting forms and finding new fields and elements through which defect triggering can be submitted to the web application. We find that dependency tracking also plays an important role in reducing the search space of web interaction sequences.

## Acknowledgment

We thank the anonymous reviewers and shepherd for their insightful feedback. We also acknowledge Emily Wan for her contributions with web application implementation and testing. Funding for this work was provided in part by, NSERC Discovery Grant RGPIN-2018-05931, a contract with Telus, and NSERC Alliance Grant ALLRP 586310-23. David Lie is supported by a Tier 1 Canada Research Chair.

## References

[1] (2021) Owasp top ten. [Online]. Available: https://owasp.org/www-project-top-ten/

[2] J. Zhu, B. Chu, H. Lipford, and T. Thomas, “Mitigating access control vulnerabilities through interactive static analysis,” in *Proceedings of the 20th ACM Symposium on Access Control Models and Technologies*, ser. SACMAT ’15. New York, NY, USA: Association for Computing Machinery, 2015, p. 199–209. [Online]. Available: https://doi.org/10.1145/2752952.2752976

[3] F. Sun, L. Xu, and Z. Su, “Static detection of access control vulnerabilities in web applications.” in *USENIX Security Symposium*, vol. 64, 2011.

[4] M. Monshizadeh, P. Naldurg, and V. N. Venkatakrishnan, “Mace: Detecting privilege escalation vulnerabilities in web applications,” in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS ’14. New York, NY, USA: Association for Computing Machinery, 2014, p. 690–701. [Online]. Available: https://doi.org/10.1145/2660267.2660337

[5] J. P. Near and D. Jackson, "Finding security bugs in web applications using a catalog of access control patterns," in *Proceedings of the 38th International Conference on Software Engineering*, ser. ICSE '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 947–958. [Online]. Available: https://doi.org/10.1145/2884781.2884836

[6] B. Eriksson, G. Pellegrino, and A. Sabelfeld, "Black widow: Blackbox data-driven web scanning," in *2021 IEEE Symposium on Security and Privacy (SP)*, 2021, pp. 1125–1142.

[7] A. Doupé, L. Cavedon, C. Kruegel, and G. Vigna, "Enemy of the state: A State-Aware Black-Box web vulnerability scanner," in *21st USENIX Security Symposium (USENIX Security 12)*. Bellevue, WA: USENIX Association, Aug. 2012, pp. 523–538. [Online]. Available: https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/doupe

[8] G. Pellegrino, C. Tschürtz, E. Bodden, and C. Rossow, "jäk: Using dynamic analysis to crawl and test modern web applications," in *Research in Attacks, Intrusions, and Defenses*, H. Bos, F. Monrose, and G. Blanc, Eds. Cham: Springer International Publishing, 2015, pp. 295–316.

[9] A. Mesbah, E. Bozdag, and A. van Deursen, "Crawling ajax by inferring user interface state changes," in *2008 Eighth International Conference on Web Engineering*, 2008, pp. 122–134.

[10] A. Doupé, M. Cova, and G. Vigna, "Why johnny can't pentest: An analysis of black-box web vulnerability scanners," in *Detection of Intrusions and Malware, and Vulnerability Assessment*, C. Kreibich and M. Jahnke, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 111–131.

[11] "Testcafe," 2023. [Online]. Available: https://testcafe.io/

[12] "Skipfish - web application security scanner," 2022. [Online]. Available: https://code.google.com/archive/p/skipfish/

[13] "Arachni," 2022. [Online]. Available: https://ecsypno.com/pages/arachni-web-application-security-scanner-framework

[14] "w3af," 2022. [Online]. Available: http://w3af.org/

[15] "Xdebug-code coverage analysis," 2023. [Online]. Available: https://xdebug.org/docs/code_coverage

[16] "php-code-coverage," 2023. [Online]. Available: https://github.com/sebastianbergmann/php-code-coverage

[17] "Coverband," 2023. [Online]. Available: https://github.com/danmayer/coverband

[18] "Cvedetails," 2024. [Online]. Available: https://www.cvedetails.com/

[19] "Xss vulnerability in hotcrp," 2024. [Online]. Available: https://github.com/kohler/hotcrp/commit/d4ffdb0ef806453c54ddca7fdda3e5c60356285c

[20] "Xss vulnerability in kanboard," 2024. [Online]. Available: https://github.com/kanboard/kanboard/commit/3824e6e9aa29017e96caae10670546db85dd9ed7

[21] S. Son, K. S. McKinley, and V. Shmatikov, "Fix me up: Repairing access-control bugs in web applications," in *Network and Distributed System Security Symposium*, 2013.

[22] A. Loo, "Automated idor discovery through stateful swagger fuzzing," 2020. [Online]. Available: https://engineeringblog.yelp.com/2020/01/automated-idor-discovery-through-stateful-swagger-fuzzing.html

[23] V. Atlidakis, P. Godefroid, and M. Polishchuk, "Restler: Stateful rest api fuzzing," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, 2019, pp. 748–758.

[24] C. Zuo, Q. Zhao, and Z. Lin, "Authscope: Towards automatic discovery of vulnerable authorizations in online services," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 799–813. [Online]. Available: https://doi.org/10.1145/3133956.3134089

[25] G. Pellegrino, M. Johns, S. Koch, M. Backes, and C. Rossow, "Deemon: Detecting csrf with dynamic analysis and property graphs," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 1757–1771. [Online]. Available: https://doi.org/10.1145/3133956.3133959

[26] G. Pellegrino and D. Balzarotti, "Toward black-box detection of logic flaws in web applications," in *NDSS 2014, Network and Distributed System Security Symposium, 23-26 February 2014, San Diego, USA*, ISOC, Ed., San Diego, 2014, iSOC. Personal use of this material is permitted. The definitive version of this paper was published in NDSS 2014, Network and Distributed System Security Symposium, 23-26 February 2014, San Diego, USA and is available at : http://dx.doi.org/10.14722/ndss.2014.23021.

[27] O. van Rooij, M. A. Charalambous, D. Kaizer, M. Papaevripides, and E. Athanasopoulos, "Webfuzz: Grey-box fuzzing for web applications," in *Computer Security – ESORICS 2021: 26th European Symposium on Research in Computer Security, Darmstadt, Germany, October 4–8, 2021, Proceedings, Part I*. Berlin, Heidelberg: Springer-Verlag, 2021, p. 152–172. [Online]. Available: https://doi.org/10.1007/978-3-030-88418-5_8

[28] F. Gauthier, B. Hassanshahi, B. Selwyn-Smith, T. N. Mai, M. Schlüter, and M. Williams, "Backrest: A model-based feedback-driven greybox fuzzer for web applications," *ArXiv*, vol. abs/2108.08455, 2021.

[29] E. Trickel, F. Pagani, C. Zhu, L. Dresel, G. Vigna, C. Kruegel, R. Wang, T. Bao, Y. Shoshitaishvili, and A. Doupé, "Toss a Fault to Your Witcher: Applying Grey-box Coverage-Guided Mutational Fuzzing to Detect SQL and Command Injection Vulnerabilities," in *Proceedings of the IEEE Symposium on Security and Privacy*, May 2023.

[30] S. Attwood, W. Li, and R. Kharel, "Evolutionary algorithms in web security: Exploring untapped potential," in *2020 12th International Symposium on Communication Systems, Networks and Digital Signal Processing (CSNDSP)*, 2020, pp. 1–6.

[31] F. Duchene, S. Rawat, J.-L. Richier, and R. Groz, "Kameleonfuzz: Evolutionary fuzzing for black-box xss detection," in *Proceedings of the 4th ACM Conference on Data and Application Security and Privacy*, ser. CODASPY '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 37–48. [Online]. Available: https://doi.org/10.1145/2557547.2557550

[32] A. Avancini and M. Ceccato, "Security testing of web applications: A search-based approach for cross-site scripting vulnerabilities," in *2011 IEEE 11th International Working Conference on Source Code Analysis and Manipulation*, 2011, pp. 85–94.

[33] S. Lee, S. Wi, and S. Son, "Link: Black-box detection of cross-site scripting vulnerabilities using reinforcement learning," in *Proceedings of the ACM Web Conference 2022*, ser. WWW '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 743–754. [Online]. Available: https://doi.org/10.1145/3485447.3512234

[34] X. Deng, Y. Gu, B. Zheng, S. Chen, S. Stevens, B. Wang, H. Sun, and Y. Su, "Mind2web: Towards a generalist agent for the web," 2023.

## APPENDIX

Table VII presents the absolute value of the coverage results of EvoCrawl when compared with other crawlers.

For the coverage experiments, the keywords that are used by the crawlers to avoid crawling on user pages, configuration pages, and plugin installation pages:

```
blocked_pages:[ mode=cooki, %2Fdisable,
    modulesadmin, database, atom, profile,
     update-core, password, maintenance,
    plugin, user/1/edit, user/2/edit, user
    /3/edit, CorePluginsAdmin,
    UsersManager, page=config, people,
    roles, authentication, usermanager,
    user/user, =acl, page=extension, mode=
    cookie, edituser, r=admin%2Fsetting,
    viewpmsg, logout, signout, javascript,
     login, signin, mode=auth, atom,
    authentication, acp_board, acp_captcha
    , delete_cookies, admin%2
    Fauthentication, UserListController, r
    =ldap%2Fadmin, admin%2Fmodule, %2
    Faccount, user%2Fdelete, user%2Fedit,
    mode=reg_details, users, fct=user,
    UserModificationController,
    UserCredentialController,
    TwoFactorController]
```

TABLE VII: This Table presents the coverage results of EvoCrawl compared with other crawlers. Column A\B presents the number of unique lines executed by EvoCrawl. Column A∩B shows the number of common lines covered by both crawlers. Column B\A denotes the number of unique lines executed by the other crawler.

| Crawler | BlackWidow | | | JAK | | | CrawlJAX | | |
|---|---|---|---|---|---|---|---|---|---|
| | A\B | A ∩ B | B\A | A\B | A ∩ B | B\A | A\B | A ∩ B | B\A |
| WordPress | 57398 | 45868 | 3368 | 58721 | 44545 | 1063 | 67905 | 35361 | 673 |
| HotCRP | 10706 | 17679 | 331 | 14447 | 13938 | 131 | 11412 | 16973 | 250 |
| Dokuwiki | 4191 | 12531 | 284 | 3099 | 13623 | 44 | 8885 | 7837 | 22 |
| Drupal | 42460 | 54691 | 21437 | 15843 | 45460 | 1195 | 46545 | 14758 | 767 |
| Humhub | 10064 | 21606 | 1741 | 16392 | 15279 | 398 | 22463 | 9207 | 290 |
| ImpressCMS | 6157 | 16485 | 622 | 8418 | 14224 | 624 | 11638 | 11004 | 390 |
| Kanboard | 10130 | 5246 | 4 | 9626 | 5750 | 1193 | 10681 | 4695 | 488 |
| Opencart | 8353 | 14500 | 672 | | | | | | |
| phpBB | 21781 | 14142 | 11616 | | | | | | |
| GitLab | 10775 | 172367 | 617 | 19398 | 163744 | 419 | 14323 | 168819 | 2652 |

TABLE VIII: This Table presents the coverage results of EvoCrawl compared with EvoCrawl-nodt Column A\B presents the number of unique lines executed by EvoCrawl. Column A∩B shows the number of common lines covered by both crawlers. Column B\A denotes the number of unique lines executed by the EvoCrawl-nodt.

| Crawler | EvoCrawl-nodt | | |
|---|---|---|---|
| | A\B | A ∩ B | B\A |
| WordPress | 29676 | 73590 | 21738 |
| HotCRP | 8578 | 19807 | 627 |
| Dokuwiki | 91 | 16631 | 112 |
| Drupal | 2321 | 58982 | 1318 |
| Humhub | 576 | 31094 | 34 |
| ImpressCMS | 1039 | 21603 | 604 |
| Kanboard | 4484 | 10892 | 246 |
| Opencart | 1151 | 21675 | 59 |
| phpBB | 8926 | 17340 | 3113 |
| GitLab | 3034 | 180108 | 2131 |

The full list of the access-denied sentence used by the IVD of EvoCrawl is:

```
{
    drupal: [Access denied, not
        authorized, page not found,
        permission is required, query
        argument is invalid],
    wordpress: [Not Allowed, invalid
        nounce, has expired, page not
        found, wordpress error],
    dokuwiki: [For admins only, have
        enough right, permission denied],
    opencart: [do not have permission to
        access],
    hotcrp: [Page inaccessible, not found
        , Redirection, is-error],
    kanboard: [didn't find this
        information, Access Forbidden],
    humhub: [You are not permitted, Could
        not find requested page, Error],
    phpbb: [not allowed],
    impresscms: [not allowed],
    gitlab: [Not Found]
}
```

## Appendix A
## Artifact Appendix

### A. Description & Requirements

*1) How to access:* The artifact has been published in Zenodo: https://doi.org/10.5281/zenodo.13617803. The GitHub Repository for the same is https://anonymous.4open.science/r/evocrawl-0BF8/. The README file inside the artifact includes the necessary steps for running the Artifact on target applications.

*2) Hardware dependencies:* None.

*3) Software dependencies:* Our code only supports Linux (Ubuntu preferred) operating system. The code depends on Node v12.22.12 and npm 6.14.16, and a variety of node modules. A requirements installation instruction can be found in the README.

*4) Benchmarks:* Our codes have been evaluated on 10 web applications including: WordPress-6.1.1, Drupal-9.3.15, HotCRP-v3.0b3, Dokuwiki-2022-07-31 "Igor", ImpressCMS-1.4.4, phpBB-3.3.8, Gitlab-11.5.1, Kanboard-1.2.22, Opencart-4.0.0, and Humhub-1.12.1.

### B. Artifact Installation & Configuration

The Installation and Configuration steps for the Artifact can be found in the repository README.

### C. Experiment Workflow

The high-level workflow of the experiments is: 1) install and configure the web application (benchmark) 2) enable coverage tracking and database logging for the benchmark 3) install and configure the artifact 4) execute the artifact on the benchmark.

### D. Major Claims

- (C1): EvoCrawl achieves an average code coverage increase of 59% and outperforms BlackWidow by HTML forms with the POST method 5 times more frequently.
- (C2): EvoCrawl successfully identifies eight zero-day bugs in WordPress, HotCRP, Kanboard, ImpressCMS, and GitLab.

## *E. Evaluation*

We provide experiment instructions to demonstrate that our artifact is functional, configurable, and usable. While the instructions can partially reproduce the results, full reproduction requires conducting each experiment for 24 hours on each benchmark.

*1) Experiment (E1):* [Coverage and HTML form Experiment] [30 human minutes + 8 compute-hour]: Run the artifacts on the web applications (benchmarks) with coverage tracking and data binary log enabled. Collect the global coverage (number of lines executed) and the number of submitted HTML forms after an 8-hour execution.

*[Preparation]*

- Install the target benchmark with coverage tracking and database binary log enabled. The installation guide and Dockerfile for benchmarks can be found within the *experiments/* folder of the repository.
- Follow the web application default configuration process provided by the web application, and Register an admin user on the web application instance.
- Complete the requirements of the Artifacts by updating the login credentials of the web application in the configuration files within the Artifacts. Detailed steps can be found in the root README file of the repository.

*[Execution]* Run the *crawl/monit.py* with *MODE* parameter set to *crawler*. The *monit.py* script will start all the crawler processes and shut them down after 8 hours (this duration can be changed). Detailed steps can be found in the *Crawler* section of the root README file in the repository.

*[Results]* The README file within the *experiments/* folder contains instructions on collecting the coverage and number of submitted forms from the artifact on the benchmark after the experiment.

*2) Experiment (E2):* [XSS Vulnerability Detection Experiment] [30 human minutes + 8 compute hours]: Run the artifacts on the web applications (benchmarks) with coverage tracking and data binary log enabled. Collect the number of XSS vulnerabilities detected after an 8-hour execution

*[Preparation]* Same as Experiment E1.

*[Execution]* Run the *crawl/monit.py* with *MODE* parameter set to *XSS*. The rest are the same as Experiment E1.

*[Results]* Within the *data/[target_benchmark]* folder of the repository, the *sources.json* and the *sinks.json* files should list the sources and the sinks of the XSS vulnerabilities. Matching between sources and sinks can be achieved using unique identifiers assigned by the artifact. The original experiment in the paper lasted for 24 hours, so 8 hours may not be enough to expose all the bugs.

*3) Experiment (E3):* [IDOR Vulnerability Detection Experiment] [30 human minutes + 8 compute-hours]: Run the artifacts on the web applications (benchmarks) with coverage tracking and data binary log enabled. Collect the number of IDOR vulnerabilities detected after an 8-hour execution

*[Preparation]* Same with Experiment E1 but need to register two additional users with lower-level privileges on the web application instance. The configuration file within the artifact also needs to be updated. Detailed steps for updating configurations can be found in the root README file of the repository.

*[Execution]* Run the *crawl/monit.py* with *MODE* parameter set to *IDOR*. The rest are the same as Experiment E1.

*[Results]* Within the repository, run the script *detect.sh*. The script will list all the vulnerable endpoints for IDOR vulnerabilities.