Gelato: Feedback-driven and Guided Security Analysis of Client-side Web Applications

1st Behnaz Hassanshahi
behnaz.hassanshahi@oracle.com
Oracle Labs Australia

2nd Hyunjun Lee
hyunjun.l.lee@oracle.com
Oracle Labs Australia

3rd Paddy Krishnan
Paddy.Krishnan@oracle.com
Oracle Labs Australia

***Abstract*—Modern web applications are getting more sophisticated by using frameworks that make development easy, but pose challenges for security analysis tools. New analysis techniques are needed to handle such frameworks that grow in number and popularity. In this paper, we describe GELATO that addresses the most crucial challenges for a security-aware client-side analysis of highly dynamic web applications. In particular, we use a feedback-driven and state-aware crawler that is able to analyze complex framework-based applications automatically, and is guided to maximize coverage of security-sensitive parts of the program. Moreover, we propose a new lightweight client-side taint analysis that outperforms the state-of-the-art tools, requires no modification to browsers, and reports non-trivial taint flows on modern JavaScript applications. GELATO reports vulnerabilities with higher accuracy than existing tools and achieves significantly better coverage on 12 applications of which three are used in production.**

***Index Terms*—web security, JavaScript, program analysis**

I. INTRODUCTION

The powerful and rich features of modern browsers have enabled developers to use languages, such as JavaScript, HTML, and CSS, to implement complex and highly interactive functionalities on the client side including single-page applications. A recent developer survey [37] shows that applications are increasingly built using complex frameworks, such as AngularJS [1] and React [34], revealing the importance for security analysis tools to handle such complexities. While attackers have been actively exploiting client-side vulnerabilities, such as DOM-based XSS [29], the existing tools are not effective in detecting them because they are unable to handle the inherent complexity in JavaScript applications and frameworks, which introduce additional abstraction layers that make it difficult to devise automated testing tools. Furthermore, the source of the client-side program that runs in the browsers is available to attackers. From an attacker's point of view, the client side can reveal invaluable information about the server side, such as REST end points, validation routines, and database queries. BackREST [48] is an example server-side fuzzer that uses the REST endpoints discovered by GELATO to detect zero-day vulnerabilities.

In this paper, we propose GELATO to take advantage of the unrestricted access of the source code on the client side to detect vulnerabilities both on the client and server side of web applications without requiring the server side code. To understand the problem better, we enumerate the crucial features a security-aware client-side analysis should support and report on the status of existing tools. We limit our study to dynamic analysis tools because they are better suited for highly dynamic JavaScript applications. Towards comparing the different tools, we have identified the following features.
Crawling features. F1 : Generates data inputs; **F2** : Generates event sequences; and **F3**: Focuses on detecting hyperlinks as well as improving coverage of executed JavaScript code.
Coverage and security features. F4: Supports goal-based (e.g., coverage) priority for triggering events; **F5**: Generates inputs that can satisfy required boolean guards; **F6**: Directs towards specific locations for more efficient security analysis; **F7**: Triggers specific functionality; **F8**: Processes dynamically registered event handlers; and **F9**: Supports modern frameworks such as React [34], and Knockout.js [26].

For security-related tools we also consider the ability to detect client-side vulnerabilities, such as DOM-based XSS. Also note that we do not consider tools that focus *purely* on analyzing the server's state; but we consider tools like Artemis+SID [51], which support *both* client-side and server-side analysis. Tables I and II summarize the features of the existing tools and compares them against GELATO. From a security perspective, GELATO is the only tool that supports the three features that help detect issues like DOM-based XSS in a variety of applications.

From a crawling coverage perspective most of the crawlers that focus on improving the coverage (lines of JavaScript code or number of discovered URLs) are not guided towards specific locations. Such guidance is essential in practice for security analysis to reach security-sensitive locations (sinks) efficiently. Moreover, GELATO is the only tool that leverages hybrid analysis to handle complex frameworks. Our evaluation on real-world applications in Sec. V shows the effectiveness of this technique for such frameworks.

The lack of suitable crawlers limits the state-of-the-art security analysis tools [56], [65] to report vulnerabilities that are present in the initial page(s) only, missing the vulnerabilities in the other parts of the application, as shown in Sec. V. The existing DOM-based XSS analysis tools do not have a state-aware crawler and are often installed as extensions in the browser, requiring manual interaction from the user to reach security sensitive locations [64]. While these tools can be useful as a last-resort mitigation solution in production, they are not suitable for detecting vulnerabilities during the testing

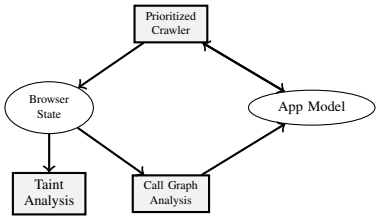

Fig. 1: High-Level Architecture

TABLE I: Comparing coverage and security features of the state-aware crawlers. The features that are only supported by GELATO are marked as red.

Feature	GELATO	Crawljax [58]	jÄk [66]	Feedex [60]	WATEG [70]	Artemis+SID [51]	Artform [69]
F1	✓	✗	✗	✗	✗	✗	✓
F4	✓	✗	✓	✓	✓	✓	✓
F5	✓	✗	✗	✗	✗	✗	✓
F6	✓	✗	✗	✗	✗	✗	✗
F7	✗	✗	✗	✓	✓	✗	✓
F8	✓	✗	✓	✗	✗	✗	✗
F9	✓	✗	✗	✗	✗	✗	✗

stages. Our contribution is integrating a crawler and a security analysis tool in a feedback-driven and guided fashion, enhancing both coverage and security analyses of highly dynamic framework-based client-side JavaScript programs. Moreover, we demonstrate a lightweight taint analysis technique that fits well into this design. The overall architecture of GELATO is shown in Fig. 1, which is described next.

Guided state-aware crawling. One of the challenges that state-aware crawlers face is that the search space they need to explore can grow exponentially. However, not all of these states need to be explored for a security analysis. To cut down the search space, our key insight is to guide a crawler to explore the relevant states. Towards providing such guidance, as shown in Fig. 1, we develop a hybrid analysis that (1) retrieves a coarse-grained model of the web application, App model, in the form of a call graph for the JavaScript code using static analysis; and (2) refines the call graph model in a feedback-driven way using execution traces generated by the crawler at runtime. This approach helps to discover new states and prioritizing events in existing states to direct the execution towards program locations of interest, such as DOM-based XSS sinks and REST calls that can be further analyzed using taint analysis. By guiding GELATO towards specific targets, we improve the performance while achieving high coverage. Moreover, by generating inputs as both data values and events, our crawler can satisfy validation routines and reach deeper parts of the application. Using static analysis to guide a fuzzer is already explored for C/C++ programs and smart contracts[43], [62]. To the best of our knowledge we are the first to take this technique to complex JavaScript applications that are heavily dynamic and event-driven. Our detailed experiments show the benefit of this technique and the opportunities to improve existing static call graph construction tools to assist dynamic analysis tools.

Staged taint inference for DOM-based XSS analysis. Once GELATO identifies the sinks and guides the crawler to reach them, it runs the security analyses. In this work, we focus on DOM-based XSS and REST endpoint discovery that is essential for server-side REST API fuzzing [48], [66], [55]. Because the DOM-based XSS analysis requires a practical dynamic taint analysis to work on modern real-world applications, we have designed a novel lightweight staged taint inference analysis that achieves better precision and recall than the state-of-the-art dynamic taint analyses [47], [65], [68], [44], [56], [57]. Moreover, our instrumentation is non-intrusive and is unlikely to break the semantics of the applications. In summary, we make the following contributions:

- A new crawler that can be guided towards program locations of interest commencing from a statically constructed call graph that is continuously refined based on execution traces.
- A feedback-driven analysis that enables our guided crawler to support modern client-side JavaScript libraries and frameworks.
- A staged taint inference analysis that detects potential DOM-based XSS vulnerabilities with high accuracy.
- A lightweight input generator that supports both event and data value generation to increase the coverage of security analyses.
- A comparison of the state-of-the-art crawlers on a wide range of applications that use complex frameworks.

Feature	GELATO	jÄk [66]	DexterJS [65]	Kudzu [67]	CTT [56]
F1	✓	✗	✓	✓	✗
F2	✓	✓	✓	✗	✗
F3	✓	✓	✗	✓	✗

TABLE II: Comparing crawling features of the client-side security analysis tools.

II. RELATED WORK

Client-side web application crawling. Crawljax [58] is a state-aware crawler that explores AJAX-based applications by comparing the states using the edit distance of the string representation of DOM trees. jÄk [66] and Black Widow [45] analyze web applications by handling dynamically generated URLs, dynamic event registration, etc., and by increasing the coverage of client-side program, they aim to find more vulnerabilities on the server side. FEEDEX [60] uses a state-aware feedback-directed crawling technique to derive a test model for client-side web applications. The main focus of FEEDEX is to reduce the test model size and enhance coverage with respect to functionality, navigation, and page structure. Unlike GELATO, none of these crawlers guide the execution towards security-sensitive sinks in the JavaScript code, such as AJAX calls for REST API testing, and DOM manipulation calls for DOM-XSS detection.

Guided crawling aims to achieve a particular goal in exploration, such as increasing code, functionality or navigation coverage [41], [60], [70]. [42] guides the exploration to discover as many states as possible in a given amount of time. Compared to [41], [60], [70], which focus on increasing the diversity of crawled pages, this work mainly focuses on increasing efficiency for an anticipated model. GELATO is also guided and proposes prioritization strategies but aims at increasing coverage of specific security-sensitive sinks.

Input value generation. In addition to event-based inputs (e.g., clicking), client-side web applications also accept input

value in URLs and form elements, such as input fields. Therefore, to have an in-depth exploration of a client-side application, input value generation is required. For input fields, existing approaches mostly provide random data if no custom data is available [59]. Tools that rely on heavyweight analyses [67], [69], such as symbolic execution, are known to have scalability issues. Furthermore, Kudzu [67] does not consider the event-driven nature of modern applications and the effort expended in input generation using symbolic execution does not have the desired consequence of increased coverage.

Taint analysis. Several static analysis techniques have been proposed to analyze JavaScript applications [52], [50], [54], [63]. However, the lack of static typing in JavaScript as well as the asynchronous and event-based nature of web applications makes it hard to detect taint flows statically. On the other hand, dynamic analysis requires instrumentation, which can be achieved at engine [56], [71], [57] or source-code level [68], [65], [23], [47], [44], [53]. Engine-level instrumentation involves adding hooks to the JavaScript engine. While this design can have performance benefits from being compiled into the engine itself, it is not easy to maintain across different engines [56]. On the other hand, source code-level instrumentation [18], [44], [65], [53] involves modifying the program's source code without affecting the original behavior, which can have lower performance, but is often easier to maintain and engine-agnostic.

DexterJS [65] carries out character-level taint tracking using code-rewriting to discover potentially vulnerable taint flows. It attaches taint labels to primitive values by wrapping (boxing) them. Because built-ins, browser APIs and DOM functions cannot be instrumented, it requires hard-coded models, which are very challenging to create. Our experiments with DexterJS [65] show that incomplete models for built-ins result in missing valid taint flows. Jalangi2 [18] provides syntactic traps to implement dynamic analyses. Our taint inference analysis uses the source code-level instrumentation in Jalangi2. Note that we only use Jalangi2 to embed our taint inference analysis to run at runtime. Our analysis is lightweight and does not make any intrusive changes, such as boxing primitive value and allows us to deal with the non-instrumented parts such as built-ins, browser APIs, and DOM functions. Affogato [47] is an instrumentation-based dynamic taint inference analysis tool Node.js applications. It finds injection vulnerabilities on the server side by detecting flows of data from untrusted security-sensitive sources to sinks using grey-box taint inference analysis. GELATO goes one step further and improves the precision by introducing a multi-staged approach.

III. APPROACH

GELATO automates guiding of state-aware crawling based on the requirements of a target security analysis using a feedback-driven approach. Algorithm 1 shows the main loop of our crawler. It takes as input the URL of a web application, a target security analysis, TA, and a set of function signatures and object property names, Loc, that the crawler should be guided towards (sinks). For instance, if the chosen target analysis is REST endpoint detection for server-side fuzzing, the target function signatures will include AJAX functions (e.g., `XMLHttpRequest.send`) in the JavaScript code and our crawler is guided to maximize the coverage of such function calls. The loop continues until the crawler reaches a fixpoint and there are no more new states to visit. The results of the target analysis ($Results$) is reported as output.

At each iteration, we run a target security analysis, TA, over the JavaScript and HTML code in the given state, S, and collect results. At the same time, we generate an Approximate Call Graph (ACG) using [46] on the newly discovered JavaScript code, and collect the execution trace using lightweight instrumentation. The execution trace helps us to add the newly discovered edges to the ACG as feedback, which contains the call graph for the explored parts of the application. $prioritizeEvent$ determines which event should be triggered next based on the metrics computed using ACG, the current state S, and program locations of interest Loc. We also reprioritize the events based on the feedback from the execution when ACG reports a false positive path from an event handler to a target location. In Sec. IV, we describe a novel taint inference analysis to detect DOM-based XSS vulnerabilities as an example security analysis that fits well in our approach.

Algorithm 1 Feedback-driven and guided security-aware crawler

1: inputs: web application URL, target analysis TA, target program locations Loc
2: output: $Results$
3: $ACG \leftarrow \emptyset$
4: $browser.goto(URL)$
5: **while** $browser.newStateExists()$ **do**
6: $S \leftarrow browser.getNewState()$
7: $Results \leftarrow$ ANALYZE(S, TA) if Loc in S // See Algorithm 2
8: $cg \leftarrow computeACG(S)$
9: $trace \leftarrow getExecutionTrace(S)$
10: $ACG \leftarrow refineACG(ACG, cg, trace)$
11: $e \leftarrow prioritizeEvent(S, ACG, Loc, trace)$
12: $browser.trigger(e)$
13: **end while**
14: $report(Results)$

State representation and comparison. The first step in designing a state-aware crawler is to define a suitable state representation. Ideally, storing the entire browser and server states will help transitioning between relevant states. But that is unrealistic. We use the state representation and comparison strategies descibed in prior work [58] and reduce the size of states by storing the most crucial elements, such as the URL, and the DOM tree. We use depth first search (DFS) to traverse the state graph, and record references to the parent and child states to replay the sequence of events to reach the current state. When DOM elements are shown or hidden using style attributes and the DOM tree does not change, our state representation will not capture the changes. We plan to deal with such cases in a future work.

Call graph refinement. One of the contributions of GELATO is using the call graph to guide the state exploration of the crawler. The function $computeACG$ in Algorithm 1 statically generates an approximate call graph using ACG [46] for the

```
1  <!DOCTYPE html>
2  <html lang='en'>
3  <script src='./knockout-min.js'></script>
4  <script>
5  function event_handler() {
6      fetch('http://localhost/attack_target');
7  }
8  </script>
9  <body>
10     <button id='b_1' data-bind='click:
       ↪  button_click'>button 1</button>
11     <button id='b_2' data-bind='click:
       ↪  button_click'>button 2</button>
12     <script>
13     function KnockoutViewModel() {
14         this.button_click = function() {
15             event_handler();
16         };
17     }
18     ko.applyBindings(new KnockoutViewModel());
19     </script>
20 </body>
21 </html>
```

Listing 1: An example framework code that ACG does not analyze soundly, and misses a critical call graph edge. Our call graph refinement approach, however, is able to detect and add it to the call graph.

JavaScript code obtained from the state S. As the crawler interacts with the application's user interface, we log the function invocations in the execution trace. This execution trace is processed to determine whether an edge in the model call graph is missing. All missing edges are added to the call graph. An edge from node a to node b is missing if the call graph does not include such an edge but the execution trace does. Listing 1 is a code-snippet from the Knockout.js [26] framework. Due to a complex event delegation mechanism, ACG fails to find the edge from the `click` event in the two `button` elements to `event_handler` function. However, once the crawler clicks on one of the buttons, e.g., button 1 at line 10, the execution trace records the edge from click event handlers in Knockout.js and `event_handler` function, which is added to the call graph. This enables us to prioritize clicking on button 2 at line 11 in future to trigger the HTTP endpoint at line 6, as described next. Note that while we are able to collect execution trace for libraries and frameworks (e.g., Knockout.js, React, jQuery [26], [34], [20]), due to limitations of source-code instrumentation, we do not trace the execution through the browser runtime and JavaScript engine. Moreover, our trace collection has limitations in supporting hard-to-analyze JavaScript language features, including (but not limited to) asynchronous and reflective call mechanisms (e.g.,`Promise` and `Function.prototype.call`).

Prioritization. We prioritize a state that is visited for the first time (not similar to any of the previously visited states) if it contains a target program location (Loc in Algorithm 1). For partially expanded states, we use a prioritization heuristic to choose the next event in the state that should be triggered by the browser. To guide the crawler towards target program locations, we prioritize an event if it has the minimum distance from the handler (registered to handle it) to a target location in the call graph. Because false positive edges in ACG can cause the crawler to get trapped in local optima, we use the collected execution trace, which contains the invoked function calls, as feedback to reprioritize the events as follows: if a reported ACG edge from an event handler to a target method is not observed in the execution trace, we give a lower priority to the event to give opportunities to other events to be triggered. When ACG does not report any reachable target location in a state, we randomly choose an event. Moreover, we propose a ***Composite*** mode, where both random and ACG-based distance prioritizations are combined using different weights to avoid local optimums.

$$score = p * ACG(e) + q * \texttt{rand()}$$

where p and q are the weights given to the priorities computed using ACG distance metric and a random score. By default we use 0.25, and 0.75 for p and q, respectively. Our evaluation in Section V shows that this prioritization scheme can be effective for some applications (e.g., Jenkins [19]) for which ACG does not achieve high precision.

Input value generation. While the state-aware crawler interacts with the JavaScript application, we analyze (line 7 in Algorithm 1) the JavaScript code returned from the server side[1], generating input values to bypass guards and increase coverage.[2] The input value generation is performed on a state if its code contains a target program location, hence the target analysis analyzes it (TA in Algorithm 1). Algorithm 2 shows how we generate input values.

Our input value generation technique can be thought of as a simplified and practical concolic testing [49], where path constraints are only collected for certain operations based on our experience from analyzing real-world JavaScript applications. Form input generation by itself is an extremely challenging topic and our solution provides a pragmatic approach to partially solve the problem in practice only. At high level, to bypass the guards on the execution path we collect runtime values of interest during the execution, construct path constraints, and solve them to generate inputs ($generateNewURLs$ at line 18). The guards that we aim to bypass are validation routines that must be satisfied to let the analyzer reach the deeper parts of the program. Example runtime values of interest are operands in conditional statements (e.g., `if` statement) and arguments in `string` function calls (e.g., the `string.substring` built-in function) that are triggered on the execution path. We use such logged values in constraint generation if they are tainted (the taint analysis is explained in detail in Sec. IV). These constraints are used to replace tainted characters of input values that are compared in a conditional statement.

We explain the input value generation in Algorithm 2 via an example. Listing 2 shows a simplified application that has a DOM-based XSS vulnerability [29]. In this example, a value obtained from the URL at line 12 is written to the `document` object at line 14, which modifies the DOM and allows the at-

[1]The JavaScript code is stored in the crawler state.

[2]There are several ways to provide input values into a client-side JavaScript application: forms, URLs, cookies, local storage, etc. In this paper, we show how we integrate input value generation for URLs to our state-aware crawler for simplicity. However, the same algorithm can be used for other sources of input values.

```
1  <!DOCTYPE html>
2  <html>
3  <body>
4  <button id="button1" onclick=safe()> Button 1 </button>
5  <button id="button2" onclick=unsafe()> Button 2 </button>
6  <textarea id = 'text1' style="display:none;">This is
     ↪  safe! </textarea>
7   <script>
8     function safe(){
9       document.getElementById("text1").style =
   ↪   "display:true;";
10    }
11    function unsafe(){
12      var loc = document.location.hash  //url is
   ↪   "http://example.com#action"
13      if (loc.indexOf("show")!=-1) {
14        document.write("You are visiting : " + loc +
   ↪   ".");
15      }
16    }
17  </script>
18 </body>
19 </html>
```

Listing 2: An example of HTML/JavaScript code with constraints on input value.

tacker's malicious payload to run. However, the original URL used to load the page is `"http://example.com#action"`, which does not contain `"show"`. Therefore, line 14 is not executed when the original input value is used. Next, we show how we generate an input (URL) that bypasses the validation at line 13 and allows the execution to reach line 14.

Initially, the execution path (π) in Algorithm 2 is empty and the test input queue, $InputQ$, contains the original URL. Our input generator continues generating new test inputs until $InputQ$ is empty. In each iteration, an input is removed from $InputQ$ and passed to the $runTargetAnalysis$ function, which runs the target analysis (TA) determined by the analyst. Before running the target analysis if the state has changed, we take the browser to state S by obtaining and triggering the corresponding event sequence ($eventSeq(S)$).

As the target analysis is performed at line 7 and results are added to $Results$, the conditional statements (e.g., `if` statements) are logged in π, which are used to generate path constraints and new test inputs at line 8. Going back to the example in Listing 2, when analysis executes line 13, we record the `if` statement together with the following runtime values in π: `"show"`, `-1` and `"http://example.com#action"` (value of `loc` variable). The $InputValueGen$ function in Algorithm 2 generates new inputs using the values recorded on the execution path, π. If a value in a conditional statement or `string` function call is identified by taint analysis to be tainted (See Sec. IV for details), the tainted characters and the value that they are compared against are recorded in the $Constraints$ map. For instance, the value of `loc` at line 13 in Listing 2 is inferred to be tainted by taint analysis, and the tainted characters are `"action"`. Therefore, `"action"` is added to $Constraints[3].taintedVal$ at line 15 in Algorithm 2. We also record `"show"`, which is then compared against the tainted value.

Finally, the $genConstraint$ function at line 16 in the algorithm generates the constraint `loc == "show"` which is stored in the $Constraints$ map. The $generateNewURLs$ function at line 18 replaces `"action"` with `"show"` in the original URL[3] and generates a new input[4]. The new URL is loaded, line 14 in Listing 2 is executed, and if the target analysis is DOM-based XSS detection, line 7 in Algorithm 2 reports a DOM-based XSS vulnerability.

```
1  var tmp = document.location.hash; // tmp = "#payload"
2  tmp = tmp.substring(3,7); // tmp = "yloa"
3  tmp = tmp + "123"; // tmp = "yloa123"
4  document.write(tmp.substring(4)); // "123" is written to
     ↪  DOM and is not tainted
5  document.write(tmp); // "yloa123" is written to DOM and
     ↪  "yload" is tainted
```

Listing 3: Example JavaScript program that contains a vulnerable taint flow.

Algorithm 2 Input value generation

```
1: function ANALYZE(S, TA)
2:     π ← ∅ // JavaScript execution path
3:     InputQ ← URL //initial seed input value
4:     while InputQ.isNotEmpty() do
5:         v ← InputQ.pop()
6:         browser.trigger(eventSeq(S))
7:         (π, Results) ← runTargetAnalysis(v, TA)
8:         InputQ.add(INPUTVALUEGEN(π))
9:     end while
10:    return Results
11: end function
12: function INPUTVALUEGEN(π)
13:    Constraints ← ∅
14:    for n in π if taintAnalysis(n.val) do
15:        Constraints[n.loc].taintedVal ← taintedVal(n.val)
16:        Constraints[n.loc].cons ← genConstraint(n)
17:    end for
18:    Return generateNewURLs(Constraints, URL)
19: end function
```

IV. Security Analysis: DOM-based XSS Detection

In this section, we describe a novel dynamic taint analysis to detect DOM-based XSS vulnerabilities. This security analysis is integrated into our guided crawler to drive the execution towards DOM manipulation locations, which are marked as sinks. Once the crawler reaches the states containing sinks, we perform taint analysis.

Dynamic taint analysis of JavaScript requires instrumentation either at the JavaScript engine [56], [71] or the source-code level [68], [65], [23]. An engine-level instrumentation-based analysis would require substantial maintenance effort. Moreover, it is possible for an attack to work on one engine but not another [65]. On the other hand, previous works show that a source-code level instrumentation-based approach [68], [65], [23] will face the following challenges: (1) tracking taint through non-instrumented code; and (2) attaching taint labels to primitive values because primitives cannot be extended with additional properties, hence to *wrap* primitives in an object that will have a property representing the taint label of the primitive value – aka, boxing. Empirical evidence from previous work [44], [65], [68] shows that wrapping primitives is an intrusive process and, thus, challenging to get right.

The challenges encountered by existing dynamic taint tracking solutions are explained via the example in Listing 3.

[3] `http://example.com#action`
[4] `http://example.com#show`

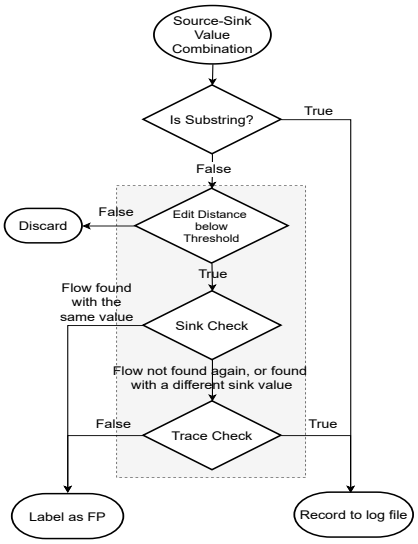

Fig. 2: Flow diagram of our staged dynamic taint flow inference technique.

It shows a JavaScript program that contains a vulnerable taint flow from the source, `location.hash` at line 1, to the security sensitive sink, `document.write` at line 5. In this example, the string value `#payload` is injected in the URL as the fragment identifier (i.e., the part of the URL following the `#` sign). DexterJS [65] fails to report the vulnerable taint flow due to an incorrect model used for the uninstrumented built-in function, `substring` at line 2. On the other hand, Linvail [44] and Chromium Taint Tracking [56] report two taint flows at both lines 4 and 5, even though `document.write` at line 4 is not tainted.

A. Dynamic Taint Inference

To address the challenges listed above, we developed a non-intrusive, dynamic taint inference analysis based on source code-level instrumentation. Our analysis infers tainted flows by correlating values at sources and sinks, and observing the behavior of the program instead of attaching and tracking taint labels. Fig. 2 shows our staged approach to discover correlations between values at taint sources and sinks. The stages, represented as diamonds in Fig. 2, act as increasingly complex filtering steps that aim to maximise the precision of our approach.

Stage 1: Substring. The first stage looks for an exact substring match of length $\geq \theta$ between string values observed at sources and sinks, i.e., whether either string is a substring of the other. If a match of length $\geq \theta$ is found, a taint flow is immediately reported. Otherwise, the remaining three stages are used to infer taint flows when the values at sources and sinks approximately match as described shortly. These stages will be described in the following sections using the symbols outlined below.

- A = a source (identified by location in source-code)
- B = a sink (identified by location in source-code)
- A_v = string value at source A
- B_v = string value at sink B
- F = a taint flow detected by our analysis

Stage 2: Edit Distance. If neither A_v nor B_v are substrings of each other (i.e., the substring stage does not report a match), the edit distance filter performs approximate matching of A_v and B_v. Specifically, this stage computes the *longest common subsequence* (LCS) [61] between A_v and B_v, extracts D_i and D_d, the number of insertions and deletions required to compute the LCS, computes a similarity score, and compares it to a threshold η. The source-sink (A, B) pairs that pass this test in this stage are recorded and processed in the next stages to filter out false positives (FPs).

Stage 3: Sink Check. In Fig. 2, when a source-sink pair (A, B) reaches the sink check stage, we know that there is no exact substring match but that the two strings are *similar*. To weed out cases where the similarity happens by chance but there is no taint flow from A to B, the sink check stage mutates A_v into A'_v by changing a few characters randomly, running the program again with the new source input, and observing B'_v. There are three possible outcomes:

1) Sink B is not reached (B'_v is `NULL`). The execution path triggered by A'_v has diverged from the execution path triggered by A_v. The pair (A, B) proceeds to the next stage.
2) B'_v is different from B_v, indicating that the value at A has an impact on the value at B. The pair (A, B) proceeds to the next stage.
3) B'_v is identical to B_v, indicating that the value at A probably has no impact on the value at B. The pair (A, B) does not proceed to the next stage.

Stage 4: Trace Check. Trace Check is the final and most expensive stage of our taint flow inference process. It aims to detect real taint flows with high precision. This step involves recording the JavaScript execution trace and analyzing the string manipulation operations performed on A_v to determine whether B_v is derived from A_v (i.e., there is a taint flow from A to B). Algorithm 3 shows our trace check stage. Given a source value A_v, a number of insertions D_i, a number of deletions D_d, and the execution trace seeded with A_v, the TRACECHECK procedure determines whether the string operations in the trace can possibly transform A_v into B_v. The sub-procedure ISOPTAINTED in Algorithm 3 re-uses the Substring and Edit Distance stages, parameterised with θ and η, to determine whether the base variable or any argument of a string operation matches A_v. If the base variable or any argument matches A_v, ISOPTAINTED returns true. The trace check stage counts the number of tainted string operations that are insertions (D_{ti}), and deletions (D_{td}) in the trace. Then, it weeds out traces where either no insertion happens while $D_i > 0$ or no deletion happens while $D_d > 0$.

We now revisit the example in Listing 3 to show how our taint inference technique correctly reports an inferred taint flow at line 5 and does not report any flows at line 4. For the sink at line 4, the observed values are A_v = `"#payload"` and B_v = `"123"`. Since A_v and B_v do not pass "Is Substring" check, they are passed to the Edit Distance stage. Because these values also fail to pass the "Edit Distance below Threshold" check, the analysis does not infer any taint flows.

For the sink at line 5 in this example, A_v = `"#payload"`, B_v = `"yloa123"`, $D_i = 3$, $D_d = 4$. Inspecting the trace

Algorithm 3 Trace Check Stage

```
1: function TRACECHECK(A_v, D_i, D_d, trace)
2:     Let D_ti = 0 and D_td = 0
3:     for each string_op in trace if ISOPTAINTED(string_op, θ, η, A_v) do
4:         if ISOPINSERTION(string_op) then
5:             D_ti += 1
6:         else if ISOPDELETION(string_op) then
7:             D_td += 1
8:         end if
9:     end for
10:    if (D_i > 0 && D_ti == 0) || (D_d > 0 && D_td == 0) then
11:        return "Trace does not match"
12:    end if
13:    return "Trace matches"
14: end function
```

between lines 1 and 4 in this example, the Trace Checking filter is able to determine that one string concatenation and two substring operations occurred. Because both of these operations are performed on the `tmp` base variable with the string values `"#payload"`, `"yloa"` and `"yloa123"`, they pass the ISOPTAINTED check at line 3 in Algorithm 3 that compares them against the source value `#payload`. The trace check algorithm then computes $D_{ti} = 1$, and $D_{td} = 2$ and concludes that the trace matches at line 13 in Algorithm 3.

V. IMPLEMENTATION AND EVALUATION

GELATO automatically interacts with the web application using the Selenium WebDriver [36]. To guide our input generator towards target locations, we use the pessimistic mode in Approximate Call Graph analysis [46] to statically build call graphs. ACG also provides an optimistic mode to improve recall by tracking more data flows. However, our experiments on real-world applications showed that it only had a small impact on recall, while degrading precision and performance. We have developed a new lightweight instrumentor to carry out the dynamic analysis of JavaScript code for dynamic event handler registration and call graph refinement. The dynamic analysis finds dynamically registered events[5], new pages, and missing edges in the call graph at runtime. The lightweight instrumentor is also used to collect runtime values and generate constraints for creating new input values. The taint inference analysis is implemented using source-code instrumentation in Jalangi2 [18] framework. Note that Jalangi2 is only an instrumentation framework and does not provide any analysis on its own. We have implemented two target security analyses in our framework, REST endpoint and DOM-based XSS detection, but our technique is not limited to these analyses and can support other security analyses too.

The experiments are performed using the Google Chrome browser version 69.0.3494.0 running on Ubuntu 16.04 VM on VirtualBox 6.0, Intel i7-7700 CPU @ 3.60GHz x 4 (4 cores assigned to VM) with 4096 MB memory. In our experiments, we study the following research questions:

- RQ1: How does GELATO compare with existing crawlers?
- RQ2: How effective are the different guiding strategies in GELATO in terms of coverage and performance?
- RQ3: Does GELATO's DOM-based XSS analysis find taint flows with better accuracy, compared to other taint analysis tools?

[5]These are the events that are registered using `addEventListener`

Benchmarks. We have gathered open-source applications, in-house applications that are currently used in production[6], vulnerable libraries, and micro-benchmarks. The advantage of such a benchmark, compared to Alexa top websites, is that we can automate authentication and test beyond the first landing page, gaining a more in-depth understanding of different crawling strategies. All the applications that are not publicly deployed and available via a URL run in Docker containers [9], which makes it easy to reset the application state and database between runs. Table III shows the selected applications and their features in our benchmarks. Our benchmark includes both single-page and multi-page applications that use modern and complex frameworks and libraries, including React, AngularJS, Knockout.js, and jQuery [34], [1], [26], [20].[7] To evaluate our feedback-driven technique on these frameworks, we have created scripts that assist the static call graph by adding the critical missing edges. These assisting scripts are about 100 lines of JavaScript code. We compare the results of our feedback-driven technique (*FD-ACG*) with these assisted versions (*Assisted-ACG*). To estimate the size of JavaScript code used in each application, we extracted the JavaScript source-code (including inline scripts) at runtime using our crawler. While we compare GELATO with other crawlers on all the applications, the guiding strategies in our crawler are compared only on the AJAX-based applications because the goal in that experiment is to evaluate the coverage of AJAX calls. Some of our open-source benchmarks are collected from [66] and the rest are selected because they use modern complex frameworks and libraries.

Application	Type	Framework/ Library	JavaScript LoC
Firing Range (FR) [14]	multi-page	-	29k
DVWA [8]	multi-page	-	218
Wivet [40]	multi-page	jQuery, Ext JS[11]	43k
WebGoat [31]✓	single-page	jQuery, AngularJS	34k
WebScanTest (WST) [39]✓	multi-page	React, AngularJS	425k
JuiceShop [24]✓	single-page	AngularJS, jQuery	20k
Jenkins [19]✓	multi-page	jQuery	52k
Apostrophe [2]✓	single-page	jQuery	139k
Keystone [25]✓	single-page	React	218k
App1✓	single-page	AngularJS, jQuery	35k
App2✓	single-page	jQuery, Knockout JS	69k
App3✓	single-page	jQuery, Knockout JS	132k

TABLE III: Benchmark Description. To count the LoC, we remove comments [38], and unminify [22] the extracted code. AJAX-based applications are marked with ✓.

A. Comparing GELATO with existing crawlers

To answer the first research question, we compare GELATO against state-of-the-art crawlers that are designed to handle dynamic web applications. The timeout used in this experiment is six hours for all applications, and we do not report the number of static URLs, such as image and JavaScript files, to focus on the crawler's ability to reach and interact with the dynamic parts of the application. We also filter query

[6]The internal applications (app1, app2, and app3) are anonymized based on the double-blind rules. Note that these applications are actively used in production, and are not created by the authors of the paper.

[7]This list only shows some candidate frameworks and libraries for each application and is not comprehensive.

TABLE IV: Comparing GELATO against other crawlers for number of URLs and AJAX calls.

Crawlers	Tested Web Applications											
	FR	DVWA	Wivet	WebGoat	App1	App2	App3	WST	JuiceShop	Jenkins	Apostrophe	Keystone
GELATO	425	69	84	409	39	20	17	307	79	279	24	70
Arachni [3]	256	17	83	6	11	1	3	25	31	61	4	17
ZAP-Crawljax [33]	83	19	15	13	8	8	1	60	30	168	5	44
Htcap [17]	160	62	68	65	7	1	13	168	15	67	18	69
jÄk [66]	257	33	76	8	20	1	-	17	2	-	6	-

parameter values in URLs because some URLs contain values, such as timestamps.[8] We use the Crawljax (AJAX Spider) plugin integrated in ZAP 2.10.0 [33] because ZAP provides better support for authentication compared to the original Crawljax implementation [5] and achieves more coverage. We use the autologin plugin in Arachni [3] for all the applications except for Jenkins, for which we inject cookies to establish the session. We also fixed database and authentication issues while evaluating Htcap [17] and jÄk [66] to the best of our ability. However, jÄk was not able to analyze three applications. We use URLs and AJAX calls as proxy to measure coverage, which are suitable for security analysis because they expose entry points to the server side. While an alternative metric would be code coverage on the client side, a lot of the JavaScript code that is executed by the browsers is not security relevant. Table IV shows the total number of URLs and AJAX calls recorded by each tool. The results show that our crawler records more URLs and AJAX calls compared to other tools. Note that GELATO outperforms other crawlers on both framework-based applications and on applications like DVWA and Firing Range that do not use any frameworks. Our results also show how other crawlers compare with each other on different types of applications. For instance, ZAP-Crawljax performs well on less modern applications, such as Jenkins, but is not as effective for most of the single-page applications. Htcap, on the other hand, is able to achieve better coverage for single-page applications, such as Apostrophe, WebGoat, App3, and Keystone.

B. Effectiveness of different guiding strategies

In the previous experiment we showed that GELATO achieves better coverage compared to the state-of-the-art crawlers. Here, we answer the second research question by taking a close look at various guiding strategies in GELATO to understand how the strategies in GELATO can affect the coverage and performance. In this experiment, we use REST endpoint discovery [66], [55] as the target security analysis. REST endpoint discovery is the key step in finding server-side vulnerabilities, such as XSS and SQLi [30], that web application fuzzers [48], [4], [32] rely on. Our goal is to measure how effective different strategies are in automatically guiding the crawler to call AJAX functions (JavaScript functions that trigger REST endpoints). We count the number of distinct AJAX calls made by each strategy over time on the AJAX-based applications shown in Table III. The timeout for all the experiments in this section is six hours.

[8]Jenkins uses object ID as path parameter in some URLs. We merge such path parameters and count them as one to avoid skewing the results.

Guiding strategies. We compare three guided and a random crawling strategies: (1) Assisted-ACG, which uses manually crafted scripts to assist finding missing edges for libraries, such as jQuery, when the approximate call graph fails to analyze them effectively; (2) FD-ACG, which is our feedback-driven technique to add newly found call graph edges at runtime automatically; and (3) Composite, which combines both Random and ACG-based distance prioritization as shown in Sec. III, as well as the assisting scripts and the feedback-driven technique (similar to the first two modes). Note that the Random strategy still benefits from features, such as dynamic event registration, and replaces prioritization functions in Algorithm 1 with random selection only.

To identify which guiding strategy suits a specific type of application, Fig. 3 compares the percentage of AJAX calls triggered by each mode over the total number of AJAX calls found by GELATO across all runs. Because we have a black-box view of the server side, this total number estimates an approximate coverage, and may not include all the AJAX calls. We have run GELATO *three* times in each mode and computed the average number across runs. The results show that our guiding strategies achieve better coverage compared to a non-guided crawling (Random), except for Keystone and WebScanTest. Keystone uses React [34] and ACG call graphs have low precision on this framework, causing our crawler to wrongly prioritize events that do not lead to AJAX calls. The ACG refinement mode for Keystone improves the coverage among the ACG-based modes by detecting true positive call graph edges during the runtime. WebScanTest performs many similar AJAX calls in one part of the application that results in ACG-based strategies spending too much time on them. We can observe some patterns for certain frameworks, such as AngularJS [1] that is used in app1, WebGoat, and Juice-Shop, where the FD-ACG mode performs almost as good as Assisted-ACG. Therefore, the assisting scripts for these applications are not crucial. However, the Assisted-ACG still performs better than FD-ACG for app2 and app3 that use the Knockout.js [26] framework. The Composite mode performs equally or better than Assisted-ACG in four applications. So, combining the ACG-based guiding strategies with random prioritization, while automatically assigning weights to each, can be an interesting direction for a future work. Finally, Apostrophe, which heavily uses jQuery benefits from the Assisted-ACG the most and FD-ACG is not effective for this application.

Fig. 4 studies the effectiveness of the guiding strategies over time. Due to space limitations, we chose four applications that use AngularJS, Knockout.js, React, and jQuery. For all of the

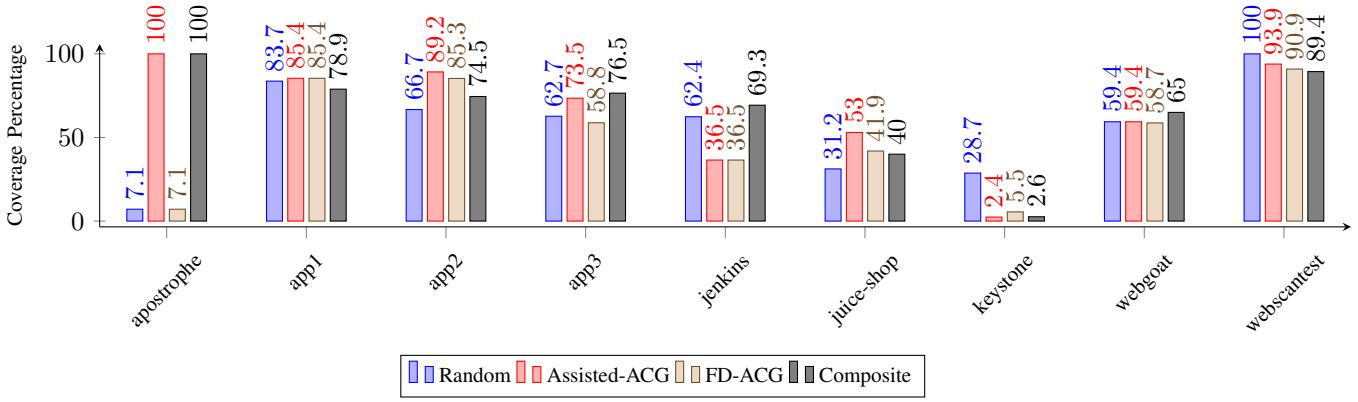

Fig. 3: Comparing overall percentage of crawling strategies over our observed ground truth.

applications, the crawler triggers a large number of AJAX functions in the first few minutes. Over time, the Random strategy performs the worst in most cases. But combining Random with ACG prioritization achieves better coverage except for Juice-Shop. The coverage gained by the ACG refinement strategy, FD-ACG, is comparable with the Assisted-ACG strategy. This is encouraging as it enables supporting complex frameworks automatically. Another observation is that even though the ACG-based guiding strategies achieve better coverage overall, the low precision of the ACG call graphs prevents GELATO to trigger the target locations fast, wasting time in triggering events that do not trigger AJAX calls for a long time. Note that the Composite mode also does not seem to solve the problem in some of the cases and more precise call graphs are crucial. Even though the quality of the static call graphs is not ideal, still getting assistance from static analysis compared to a pure dynamic analysis is beneficial due to the large performance overhead and instrumentation challenges of a dynamic analysis [47]. Overall, our results show that a guiding strategy that uses call graphs performs better than a non-guided crawler that selects events randomly in most cases (seven out of nine apps). However, different frameworks and applications require different ACG-based strategies and an initial experimental run to choose the right strategy is recommended.

C. Accuracy of staged taint flow inference

This section answers our third research question by evaluating GELATO's effectiveness for DOM-based XSS detection. First, we evaluate the effectiveness of our staged taint inference technique on microbenchmarks and libraries with known DOM-based XSS vulnerabilities. We use two open-source microbenchmarks designed to evaluate DOM-based XSS detection tools: Firing Range [14][9] from Google, and IBM benchmarks [27]. We also compare our taint inference technique against dynamic taint tracking in DexterJS [65], and CTT (Chromium Taint Tracking) [56], the state-of-the-art DOM-based XSS detection tools. We noticed that the crawling capabilities of these two tools are very limited or non-existent. Thus, in this paper, we do not compare them against GELATO's crawler. Second, we evaluate the effectiveness of our guided crawling strategy and input generation technique for DOM-based XSS detection analysis. We use 0.09 as the similarity score threshold for the Edit Distance and minimum length of two characters for the Substring check in Fig. 2.

Taint flow inference on microbenchmarks. The payloads for a DOM-based XSS attack are often provided in the URLs, which can easily be controlled by attackers. While some of the test cases in the Firing Range and IBM benchmarks contain a valid flow from a source to a sink, the value at the source cannot be directly tainted through URLs, e.g., `sessionStorage`. The TP (True Positive) test cases in Table V consider only cases that can be triggered through a URL (URL-controllable), while the FP (False Positive) cases contain sources that are non URL-controllable or infeasible taint flows. Note that because our taint inference technique can track taint values passed via browser APIs without requiring to model them, we expect to have better precision than taint tracking techniques that rely on models for non URL-controllable cases. We plan to extend our implementation to support non URL-controllable taint sources in future.

Benchmark	Firing Range			IBM
	address tests	urldom tests	dom tests (toxicdom)	
	R / P	R / P	R / P	R / P
# GELATO	82 / 100	85 / 100	100 / 100	90 / 100
# DexterJS	75 / 100	23 / 100	100 / 10	80 / 77
# CTT	64 / 100	38 / 100	100 / 20	82 / 75

TABLE V: Taint flow recall (R) and precision (P) reports for microbenchmarks

Because we have the full ground truth for these microbenchmarks, we can compute the recall and precision. Table V shows that GELATO finds more taint flows than DexterJS and CTT. This is particularly apparent in urldom test cases, where we detect a significantly higher portion of the URL-controllable taint flows. In particular, we can report the taint flow in non-trivial test case[10] in the IBM benchmark, which is triggered if the URL has `topic=` query parameter. GELATO successfully

[9]For the Firing Range benchmark, we evaluate only against the DOM-related test cases: address tests [12], urldom tests [15] and dom tests (toxicdom) [13].

[10]`Incorrect_Sanitizer/apollo_test_01.html in [27].`

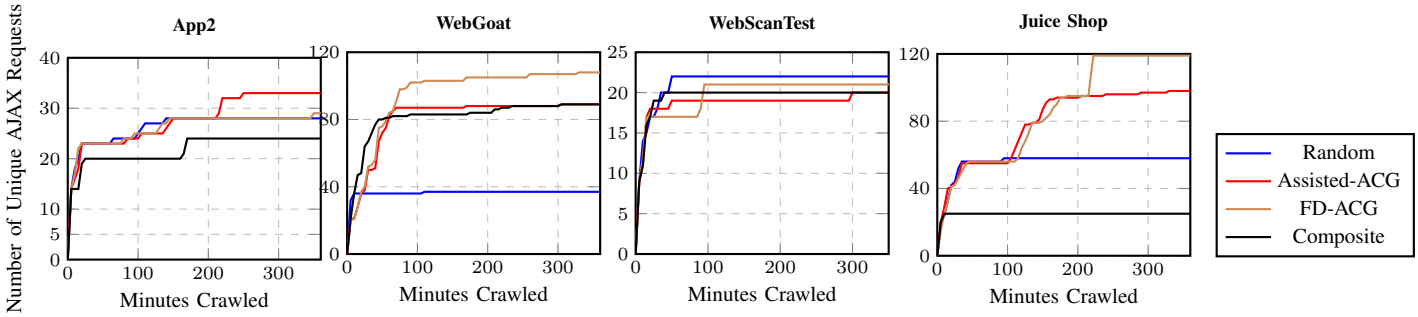

Fig. 4: Evaluating guided crawling against random crawling.

finds this query parameter and reports the taint flow. This test case shows the effectiveness of our value input generator and constraint heuristics explained in Sec.III. We investigated the results from other tools further to understand the root causes of their failures. We noticed that DexterJS does not handle some of the built-in functions, sanitizations and browser APIs. On the other hand, CTT does not handle JavaScript property reads and writes, which result in false negatives. We also noticed problems in the event-generation component of DexterJS that leads to poor coverage and missing valid taint flows. Because CTT does not have support for event generation, it misses all the flows that require user interaction. GELATO has better or equal precision than DexterJS and CTT thanks to the Sink Check and Trace Check stages in Sec. IV. Our manual inspection shows that Sink Check is enough to avoid most of the false positives but Trace Check is also used sometimes to achieve a 100% precision.

JavaScript libraries and open-source applications. Table VI reports the effectiveness of our taint inference mechanism on JavaScript libraries that have known vulnerabilities, as reported in RetireJS [35].[11] We have created test harnesses for these libraries to trigger the DOM-based XSS vulnerable paths and input payloads are provided for all the tools. GELATO is able to report taint flows for all of these vulnerable libraries while DexterJS misses all of them and CTT reports four. Dojo is an interesting library in our benchmarks because its vulnerability can be found by our constraint heuristics explained in Sec.III by bypassing the input validation that expects `theme` as the query parameter in the URL [10].

Table VI also shows that GELATO can successfully detect DOM-based XSS vulnerabilities in non-trivial modern web-applications, while the other tools miss reporting them. To detect the vulnerabilities in these applications, a state-aware crawler is needed that triggers the vulnerable path. In this experiment, GELATO is the only tool that has an effective state-aware crawler, successfully exploring the application and triggering the vulnerable path. It is worth noting that for real applications it is very challenging to get the ground truth. That is why we compare GELATO with existing tools and report only relative recall and precision. We also compared the effectiveness of guided crawling strategy for DOM-based XSS detection analysis with the Random strategy. The target locations in this experiment were DOM manipulation operations in the program. For both Firing Range and IBM benchmarks, the guided crawling strategy helps find the DOM-based XSS vulnerabilities in 50% less time than a Random strategy.

Benchmark	GELATO	DexterJS	CTT
jQuery 1.11.1 [7]	✓	×	×
jQuery 1.6.1 [6]	✓	×	✓
jQuery-migrate 1.1.1 [21]	✓	×	✓
handlebars 1.0.0.beta.2 [16]	✓	×	✓
mustache 0.3.0 [28]	✓	×	✓
dojo 1.4.1 [10]	✓	×	×
Juice-shop 8.3.0 [24]	✓	×	×
Damn Vulnerable Web App (DVWA) [8]	✓	×	×

TABLE VI: DOM-based XSS detection for libraries and open-source applications.

Threats to validity. While we aimed to select representative applications and benchmarks that use modern technologies, the choice of benchmarks might have affected the validity of the experiments presented in this paper. Moreover, obtaining ground truth for coverage of real applications is very challenging and would need to manually interact with the UI of the client-side app to capture **all** the URLs and REST endpoints. That is why we have used a relative ground truth for RQ2 (Fig. 3) based on the collective observations across all the strategies. Finally, in our experiments we showed that the DOM-based XSS detection in GELATO has a high accuracy for the analyzed applications and libraries. However, depending on the complexity of the taint manipulation operations in the given program, the accuracy could vary.

VI. CONCLUSION

In this paper, we have presented GELATO, a feedback-driven and guided security-aware crawler that addresses the gaps in analyzing complex framework-based JavaScript applications. We have studied and evaluated the state-of-the-art tools, and presented the most crucial features a security-aware client-side analysis should be supporting. We showed that our crawler outperforms existing crawlers by achieving better coverage, and presented various crawling strategies that suit different types of applications. We have also developed a new lightweight client-side taint analysis that reports non-trivial taint flows in modern JavaScript applications with higher accuracy than the existing dynamic taint analysis tools.

[11] RetireJS documents vulnerabilities in "retired" versions of JavaScript libraries.

REFERENCES

[1] AngularJS. https://angularjs.org/, 2021. Accessed: 2021-02-1.

[2] Apostrophe boiler-plate. https://github.com/apostrophecms/apostrophe-boilerplate, 2021. Accessed: 2021-02-1.

[3] Arachni Framework version 1.5.1-0.5.12. http://www.arachni-scanner.com/blog/tag/crawl/, 2021. Accessed: 2021-02-1.

[4] Burp Suite. https://portswigger.net/burp/, 2021. Accessed: 2021-02-1.

[5] Crawljax. https://github.com/crawljax/crawljax, 2021. Accessed: 2021-02-1.

[6] CVE-2011-4969. http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2011-4969, 2021. Accessed: 2021-02-1.

[7] CVE-2015-9251. https://nvd.nist.gov/vuln/detail/CVE-2015-9251, 2021. Accessed: 2021-02-1.

[8] Damn Vulnerable Web Application (DVWA). http://dvwa.co.uk/, 2021. Accessed: 2021-02-1.

[9] Docker container. https://www.docker.com/, 2021. Accessed: 2021-02-1.

[10] DOM-Based XSS in Dojo Toolkit SDK. https://bugs.dojotoolkit.org/ticket/10773, 2021. Accessed: 2021-02-1.

[11] Ext JS. https://www.sencha.com/products/extjs, 2021. Accessed: 2021-02-1.

[12] Firing Range: Address DOM-based XSS. https://public-firing-range.appspot.com/address/index.html, 2021. Accessed: 2021-02-1.

[13] Firing Range: DOM tests (toxicdom). https://public-firing-range.appspot.com/dom/index.html, 2021. Accessed: 2021-02-1.

[14] Firing Range Test Bed. https://public-firing-range.appspot.com/, 2021. Accessed: 2021-02-1.

[15] Firing Range: URL-based DOM-based XSS. https://public-firing-range.appspot.com/urldom/index.html, 2021. Accessed: 2021-02-1.

[16] Handlebars.js GitHub Issue #68. https://github.com/wycats/handlebars.js/pull/68, 2021. Accessed: 2021-02-1.

[17] Htcap v1.0.1. https://github.com/segment-srl/htcap/releases, 2021. Accessed: 2021-02-1.

[18] Jalangi2. https://github.com/Samsung/jalangi2, 2021. Accessed: 2021-02-1.

[19] Jenkins 2.60.3. https://github.com/jenkinsci/docker, 2021. Accessed: 2021-02-1.

[20] jQuery. https://jquery.com/, 2021. Accessed: 2021-02-1.

[21] jQuery Bug Tracker Issue #11290. https://bugs.jquery.com/ticket/11291, 2021. Accessed: 2021-02-1.

[22] JS Beautify. https://github.com/beautify-web/js-beautify, 2021. Accessed: 2021-02-1.

[23] jsTaint. https://github.com/idkwim/jsTaint, 2021. Accessed: 2021-02-1.

[24] Juice-shop 8.3.0. https://github.com/bkimminich/juice-shop, 2021. Accessed: 2021-02-1.

[25] Keystonejs v1.1. https://hub.docker.com/r/ntutselab/keystonejs, 2021. Accessed: 2021-02-1.

[26] Knockout.js. https://knockoutjs.com/, 2021. Accessed: 2021-02-1.

[27] LaBaSec: Language-based Security. http://m.ibm.com/http/researcher.ibm.com/researcher/view_group_subpage.php?id=1598, 2021. Accessed: 2021-02-1.

[28] Mustache.js GitHub Issue #112. https://github.com/janl/mustache.js/issues/112, 2021. Accessed: 2021-02-1.

[29] OWASP DOM-Based XSS. https://www.owasp.org/index.php/DOM_Based_XSS, 2021. Accessed: 2021-02-1.

[30] OWASP Top 10. https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project, 2021. Accessed: 2021-02-1.

[31] OWASP WebGoat Project. https://www.owasp.org/index.php/Category:OWASP_WebGoat_Project, 2021. Accessed: 2021-02-1.

[32] OWASP ZAP. https://www.owasp.org/index.php/OWASP_Zed_Attack_Proxy_Project, 2021. Accessed: 2021-02-1.

[33] OWASP ZAP 2.10.0. https://www.zaproxy.org, 2021. Accessed: 2021-02-1.

[34] React.js. https://reactjs.org/, 2021. Accessed: 2021-02-1.

[35] Retire.js. http://retirejs.github.io/retire.js/, 2021. Accessed: 2021-02-1.

[36] Selenium Web Driver. https://www.selenium.dev/documentation/en/webdriver/, 2021. Accessed: 2021-02-1.

[37] Stack Overflow Developer Survey. https://insights.stackoverflow.com/survey/2020#technology-web-frameworks, 2021. Accessed: 2021-02-1.

[38] Strip Comments. https://github.com/jonschlinkert/strip-comments, 2021. Accessed: 2021-02-1.

[39] WebScanTest. https://www.webscantest.com/, 2021. Accessed: 2021-02-1.

[40] Wivet. https://github.com/bedirhan/wivet, 2021. Accessed: 2021-02-1.

[41] Shay Artzi, Julian Dolby, Simon Holm Jensen, Anders Møller, and Frank Tip. A Framework for Automated Testing of JavaScript Web Applications. In *ICSE*, 2011.

[42] Kamara Benjamin, Gregor Von Bochmann, Mustafa Emre Dincturk, Guy-Vincent Jourdan, and Iosif Viorel Onut. A Strategy for Efficient Crawling of Rich Internet Applications. In *ICWE*, 2011.

[43] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. Directed Greybox Fuzzing. In *CCS*, 2017.

[44] Laurent Christophe, Elisa Gonzalez Boix, Wolfgang De Meuter, and Coen De Roover. Linvail: A general-purpose platform for shadow execution of JavaScript. In *SANER*, 2016.

[45] Benjamin Eriksson, Giancarlo Pellegrino, and Andrei Sabelfeld. Black Widow: Blackbox Data-driven Web Scanning. In *SSP*, 2021.

[46] Asger Feldthaus, Max Schäfer, Manu Sridharan, Julian Dolby, and Frank Tip. Efficient Construction of Approximate Call Graphs for JavaScript IDE Services. In *ICSE*, 2013.

[47] François Gauthier, Behnaz Hassanshahi, and Alexander Jordan. AFFOGATO: Runtime Detection of Injection Attacks for Node.Js. In *SOAP*, 2018.

[48] François Gauthier, Behnaz Hassanshahi, Benjamin Selwyn-Smith, Trong Nhan Mai, Max Schlüter, and Micah Williams. BackREST: A Model-Based Feedback-Driven Greybox Fuzzer for Web Applications. *CoRR*, 2021.

[49] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: directed automated random testing. In *PLDI*.

[50] Salvatore Guarnieri, Marco Pistoia, Omer Tripp, Julian Dolby, Stephen Teilhet, and Ryan Berg. Saving the World Wide Web from Vulnerable JavaScript. In *ISSTA*, 2011.

[51] Casper Svenning Jensen, Anders Møller, and Zhendong Su. Server interface descriptions for automated testing of JavaScript web applications. In *ESEC/FSE*, 2013.

[52] Simon Holm Jensen, Anders Møller, and Peter Thiemann. Type Analysis for JavaScript. In *SAS*, 2009.

[53] Rezwana Karim, Frank Tip, Alena Sochurková, and Koushik Sen. Platform-independent dynamic taint analysis for javascript. *IEEE Trans. Software Eng.*, pages 1364–1379, 2020.

[54] Hongki Lee, Sooncheol Won, Joonho Jin, Junhee Cho, and Sukyoung Ryu. SAFE: Formal specification and implementation of a scalable analysis framework for ECMAScript. In *FOOL*, 2012.

[55] Manuel Leithner and Dimitris E. Simos. XIEv: Dynamic Analysis for Crawling and Modeling of Web Applications. In *SAS*, 2020.

[56] Sebastian Lekies, Ben Stock, and Martin Johns. 25 million flows later: large-scale detection of DOM-based XSS. In *CCS*, 2013.

[57] William Melicher, Anupam Das, Mahmood Sharif, Lujo Bauer, and Limin Jia. Riding out DOMsday: Towards Detecting and Preventing DOM Cross-Site Scripting. In *NDSS*, 2018.

[58] Ali Mesbah, Arie van Deursen, and Stefan Lenselink. Crawling Ajax-Based Web Applications Through Dynamic Analysis of User Interface State Changes. *ACM Trans. Web*, 2012.

[59] Ali Mesbah, Arie van Deursen, and Danny Roest. Invariant-Based Automatic Testing of Modern Web Applications. *IEEE Trans. Softw. Eng.*, 2012.

[60] Amin Milani Fard and Ali Mesbah. Feedback-directed exploration of web applications to derive test models. In *ISSRE*, 2013.

[61] S Needleman and C Wunsch. A general method applicable to the search for similarities in the amino acid sequences of two proteins. In *J. Mol. Biol. 48*, 1970.

[62] Tai D. Nguyen, Long H. Pham, Jun Sun, Yun Lin, and Quang Tran Minh. sFuzz: an efficient adaptive fuzzer for solidity smart contracts. In *ICSE*, 2020.

[63] Benjamin Barslev Nielsen, Behnaz Hassanshahi, and François Gauthier. Nodest: Feedback-Driven Static Analysis of Node.Js Applications. In *ESEC/FSE*, 2019.

[64] Jinkun Pan and Xiaoguang Mao. Domxssmicro: A micro benchmark for evaluating dom-based cross-site scripting detection. In *IEEE Trustcom/BigDataSE/ISPA*, 2016.

[65] Inian Parameshwaran, Enrico Budianto, Shweta Shinde, Hung Dang, Atul Sadhu, and Prateek Saxena. Auto-patching DOM-based XSS at scale. In *FSE*, 2015.

[66] Giancarlo Pellegrino, Constantin Tschürtz, Eric Bodden, and Christian Rossow. jÄk: Using Dynamic Analysis to Crawl and Test Modern Web Applications. In *RAID*, 2015.

[67] Prateek Saxena, Devdatta Akhawe, Steve Hanna, Feng Mao, Stephen McCamant, and Dawn Song. A Symbolic Execution Framework for JavaScript. In *SP*, 2010.

[68] Koushik Sen, Swaroop Kalasapur, Tasneem Brutch, and Simon Gibbs. Jalangi: A Selective Record-replay and Dynamic Analysis Framework for JavaScript. In *ESEC/FSE*, 2013.

[69] Ben Spencer, Michael Benedikt, Anders Møller, and Franck van Breugel. ArtForm: A Tool for Exploring the Codebase of Form-based Websites. In *ISSTA*, 2017.

[70] Suresh Thummalapenta, K. Vasanta Lakshmi, Saurabh Sinha, Nishant Sinha, and Satish Chandra. Guided Test Generation for Web Applications. In *ICSE*, 2013.

[71] Philipp Vogt, Florian Nentwich, Nenad Jovanovic, Engin Kirda, Christopher Krügel, and Giovanni Vigna. Cross Site Scripting Prevention with Dynamic Data Tainting and Static Analysis. In *NDSS*, 2007.