DOM-XSS Detection via Webpage Interaction Fuzzing and URL Component Synthesis*

Nuno Sabino[†‡], Darion Cassel[‡°], Rui Abreu[§], Pedro Adão[†], Lujo Bauer[‡] and Limin Jia[‡]
[†]Instituto Superior Técnico, Universidade de Lisboa, and Instituto de Telecomunicações
[‡]Carnegie Mellon University, [§]Universidade do Porto, INESC-ID [°]Work done prior to joining Amazon
{nsabino,lbauer,liminjia}@andrew.cmu.edu, darion.cassel@gmail.com, rma@fe.up.pt, pedro.adao@tecnico.ulisboa.pt

***Abstract*—DOM-based cross-site scripting (DOM-XSS) is a prevalent form of web vulnerability. Prior work on automated detection and confirmation of such vulnerabilities at scale has several limitations. First, prior work does not interact with the page and thus misses vulnerabilities in event handlers whose execution depends on user actions. Second, prior work does not find URL components, such as GET parameters and fragment values that, when instantiated with specific keys/values, execute more code paths. To address this, we introduce SWIPE, a DOM-XSS analysis infrastructure that uses fuzzing to generate user interactions to trigger event handlers and leverages dynamic symbolic execution (DSE) to automatically synthesize URL parameters and fragments. We run SWIPE on 44,480 URLs found in pages from the Tranco top 30,000 popular domains. Compared to prior work, SWIPE's fuzzer finds 15% more vulnerabilities. Additionally, we find that a lack of parameters and fragments in URLs significantly hinders DOM-XSS detection, and show that SWIPE's DSE engine can synthesize previously unseen URL parameters and fragments that trigger 20 new vulnerabilities.**

I. Introduction

According to the National Vulnerability Database (NVD), 12.25% of all reported vulnerabilities are Cross-Site Scripting (XSS) vulnerabilities [44], which allow attackers to execute JavaScript code in a victim's browser. These vulnerabilities can lead to cookie theft and sensitive data leakage, which can be used to impersonate the victim or perform undesired actions on vulnerable websites [46]. XSS vulnerabilities come in several forms, commonly classified as reflected XSS, stored XSS and DOM-based XSS (DOM-XSS). The first two types involve server-side code that fails to properly sanitize user input before returning it to the client, while DOM-XSS occurs entirely within client-side JavaScript code. As a result, the server hosting the website is not able to block DOM-XSS exploits. Such attacks may also not be detected by the server, as the malicious payload could be in a URL fragment that is not transmitted to the server [28]. This makes DOM-XSS particularly challenging to detect and mitigate compared to other types of XSS vulnerabilities, as server-side protections like Web Application Firewalls (WAFs) are not as effective.

To measure DOM-XSS pervasiveness in the wild, prior work developed systems to analyze webpages to find and confirm DOM-XSS vulnerabilities [5, 31, 36]. They use a taint-tracking-enabled browser to detect *flows*—transfers of information from potentially attacker-controlled sources (e.g., URL) to API calls that lead to code execution (e.g., `document.write`). These flows represent potential attack vectors where attacker-controlled input might lead to code execution in the browser. However, infrastructures developed by prior work have limited effectiveness in analyzing and exploring diverse JavaScript code paths and enabling reproduction and comparison of analysis results across different methods. First, most existing approaches passively analyze the page without simulating user interactions. Many DOM-XSS vulnerabilities only manifest when event handlers are triggered by user actions such as clicks, keyboard input, or form submissions, as evidenced by this work (Sec. IV). Without triggering these events, vulnerabilities in event-driven code remain undetected. Second, prior approaches typically analyze each page based on a single URL string, without systematically exploring how different URL components, such as GET parameters and fragments (PFs), might enable new code paths that could contain vulnerabilities. Third, existing approaches do not address a key methodological challenge for fair comparison of results across different DOM-XSS detection techniques: webpages are dynamic and can change over time, leading to different results when re-analyzed. This makes it difficult to compare the effectiveness of detection approaches under identical conditions. These limitations leave significant blind spots in DOM-XSS detection, raising important research questions: ① How many vulnerabilities can only be triggered by actively interacting with web pages? ② Can we automatically discover URL parameters and fragments that trigger vulnerabilities, and how many of those vulnerabilities are new (i.e., not found by any other tested method)? ③ How do we enable fair comparisons between different DOM-XSS detection approaches given the dynamic nature of web content?

To address these questions, we present SWIPE (Simulator of Webpage Interactions and Parameter Explorer), a comprehensive DOM-XSS detection infrastructure. SWIPE

* Full version available at https://doi.org/10.1184/R1/30010783.

Network and Distributed System Security (NDSS) Symposium 2026
23-27 February 2026, San Diego, CA, USA
ISBN 979-8-9919276-8-0
https://dx.doi.org/10.14722/ndss.2026.231467
www.ndss-symposium.org

uses fuzzing to systematically simulate a wide range of user actions and trigger event handlers to detect DOM-XSS vulnerabilities contained in the handlers or in code loaded due to the execution of those handlers. Additionally, SWIPE employs dynamic symbolic execution (DSE) to automatically discover and synthesize URL components that can lead to the execution of previously unexplored code paths. Our tool builds upon and extends prior work by using an updated version of a taint-enabled Chromium browser from DOMsday [36]. Furthermore, to improve reproducibility and fair comparisons between detection approaches, SWIPE leverages web archiving methodology [9], which archives the page resources and replays them for analysis. Though archiving has been studied in the past for general web security measurements [19], to the best of our knowledge, SWIPE is the first work to incorporate web archiving technology for DOM-XSS detection, specifically to allow a fair comparison of different detection techniques.

This paper pursues two complementary goals: (1) measuring the prevalence of DOM-XSS vulnerabilities using improved methodology, and (2) evaluating the effectiveness of SWIPE. For that, we conduct a large-scale empirical evaluation on 44,480 URLs found by visiting pages on the top 30,000 domains from the Tranco list [30]. SWIPE components identified 114 unique DOM-XSS vulnerable flows in 146 pages. Our evaluation shows compelling evidence of SWIPE's effectiveness in improving DOM-XSS detection capabilities. By actively interacting with web pages, our fuzzer increases the detection of confirmed vulnerable flows by 15% compared to passive analysis techniques from prior work and is more effective in aiding DOM-XSS vulnerability detection than CrawlJax [38], an existing user interaction automation tool. This demonstrates that without effective user interaction exploration, a significant number of vulnerabilities may remain hidden. Our evaluation also reveals that systematically exploring URL parameters and fragments (PFs) significantly enhances DOM-XSS detection capabilities. We demonstrate that SWIPE's DSE engine finds new vulnerable flows by synthesizing additional PFs, and most of those PFs cannot be generated by off-the-shelf GET parameter fuzzing tools ffuf [20] and wfuzz [3]. With the help of our DSE engine, SWIPE outperforms the web scanner Wapiti [59], which also finds GET parameters. Finally, SWIPE outperforms the black-box web scanner ZAP [45].

This paper makes the following contributions:

1) We identify and address key limitations in existing DOM-XSS detection methods, particularly the lack of interaction with pages and exploration of URL components.
2) We develop a novel fuzzer to simulate a wide range of user actions. It can trigger event handlers and uncover vulnerabilities that require interaction to manifest.
3) We integrate dynamic symbolic execution into DOM-XSS detection to automatically synthesize URL parameters and fragments that lead to new execution paths and vulnerabilities.
4) We leverage web archiving to aid fair and reproducible comparison of DOM-XSS detection approaches.
5) We conduct a large-scale evaluation on 44,480 URLs, providing empirical evidence of SWIPE's effectiveness and insights into the prevalence and characteristics of DOM-XSS vulnerabilities in the wild.

We open-sourced the fuzzer, DSE and the web archive [51].

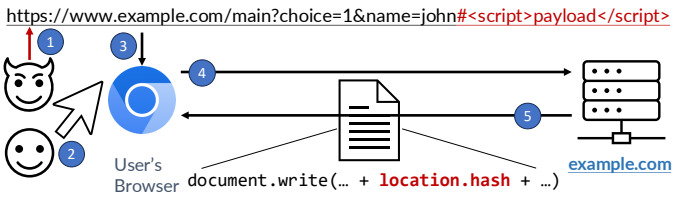


Fig. 1: Diagram illustrating a DOM-XSS attack.

II. Background

We review DOM-XSS vulnerabilities, discuss the methodologies used by prior work to detect and confirm them, and present the methodologies used by our novel DOM-XSS detection infrastructure, SWIPE. The project names of prior studies on DOM-XSS vulnerabilities are abbreviated as follows: 25mFlows [31], DOMsday [36], and TalkGen [5].

A. DOM-XSS vulnerabilities

DOM-XSS is a type of cross-site scripting attack wherein the attacker's payload is injected into the page's DOM purely as a result of client-side JavaScript code execution. The general flow of such an attack is shown in Fig. 1. First, the attacker crafts a URL to a page vulnerable to DOM-XSS (the *vulnerable page*) and adds an attacker-controlled payload (e.g., with code that sends cookies via `document.cookie` to the attacker) (①). The attacker then tricks a user (the *victim*) into visiting the URL (②). When the victim visits the URL (③), the vulnerable page is loaded (④), executing JavaScript code that allows the attacker-controlled URL components (e.g., the URL fragment accessible via `location.hash`) to reach sensitive DOM-XSS *sinks* which modify the DOM, such as `document.write` (⑤).

B. DOM-XSS detection

Most proposed techniques for detecting DOM-XSS vulnerabilities rely on dynamic taint analysis [5, 31, 36, 47]. Inputs from attacker-controllable sources (e.g., URL) are marked as *tainted*, and taint is propagated through the execution of JavaScript. When tainted data reach an API that can cause script execution, which is called a *sink*, the analysis reports a *flow*, a possible vulnerability. Examples of attacker-controllable sources are the URL, which is accessed via `document.location`; cookies; the HTTP referrer, accessed via `document.referrer`; and cross-origin messages sent via the `postMessage` API. Sinks include JavaScript-based primitives like `eval` or the `new Function` constructor and HTML-based primitives like `document.write` calls and assignments to `innerHTML` or event-handler properties such as `onclick`. The list of considered sources and DOM-XSS sinks can be found in our companion tech report [50].

25mFlows [31] used dynamic taint analysis and demonstrated the prevalence of DOM-XSS vulnerabilities in the wild with

Tainted URL	http://example.com/page?q=tainted&a=b
Prior work	Example confirmation URL
25mFlows [31]	http://example.com/page?q=tainted&a=b#PAYLOAD
DOMsday [36]	http://example.com/page?a=b#&q=PAYLOAD
TalkGen [5]	http://example.com/page?q=PAYLOAD&a=b

TABLE I: Confirmation URLs from existing methodologies.

byte-level taint tracking of JavaScript code [31]. It crawled the Alexa Top 5,000 domains and subpages and found DOM-XSS vulnerabilities on 9.6% of those domains. DOMsday [36] used a similar approach but implemented a more precise confirmation methodology (discussed below). DOMsday performed a broader but shallower crawl than 25mFlows, targeting the Alexa Top 10,000 websites with five subpages each. DOMsday found a similar prevalence of DOM-XSS potential flows but was able to confirm the vulnerability of 83% more than measured by 25mFlows. More recently, TalkGen also implemented a byte-level dynamic information flow analysis but focused on extending the confirmation methodology with targeted exploit generation [5]. It also extended the breadth of the evaluation of 25mFlows and DOMsday, crawling the Tranco top 100,000 websites at a depth of 10 subpages.

SWIPE builds off of prior work's taint analysis and confirmation methodology — but extends it with a novel user action fuzzer (Sec. III-B) and dynamic symbolic execution (Sec. III-C) to trigger execution of more webpage code.

C. DOM-XSS vulnerability validation

The presence of a flow does not necessarily imply that a vulnerability exists. This happens, for example, when the input is correctly sanitized before being passed to the vulnerable sink. A common way to sanitize inputs is to URL encode them, such that special characters (e.g., double quotes) are encoded into a safe representation, preventing the attacker from exploiting a vulnerability. A flow is considered to be *potential* by our and previous works if the source is URL-based, the sink is JavaScript or HTML-based, and if the tainted input that reaches the sink is not URL encoded.

Once a potential flow is found, all three prior works try to confirm that a vulnerability actually exists by following the recipe below: (1) Collect the URL of the potentially vulnerable resource where the flow was uncovered. (2) Find a location within that URL where the attacker payload is to be injected. This is the part of the input that eventually reaches the sink and can specify JavaScript code to execute. (3) Inject a payload at the identified location such that it allows validation of vulnerability's exploitability once the URL is visited.

A representation of the three injection methodologies from prior work is shown in Tab. I. 25mFlows implements a *breakout* method where the parsed abstract syntax tree (AST) of the HTML or JavaScript string that reaches the sink is analyzed to determine what characters are needed to complete what comes before the tainted input. A candidate is then generated and appended to the end of the potentially vulnerable page's URL, e.g., in the URL fragment (hash). DOMsday improves upon this technique by determining the exact bytes in the URL where the payload needs to be injected, based on the observation that the data reaching the sink are often a URL GET (query) parameter. TalkGen extends this by replacing tainted query parameter values by the payload, as opposed to always injecting the payload after the hash. However, injecting GET parameters *before* the hash causes these parameters to be sent to the web server, which makes testing less safe.

On step (3), DOMsday always injects the same payload *marker*<>'" instead of trying to execute real code to confirm vulnerabilities. It is important to note that the special characters on that payload (i.e., the HTML tags, the single and the double quotes) are there to ensure the attacker has the ability to escape the necessary context and execute JavaScript code. To confirm a vulnerability, DOMsday visits this synthesized URL, validates that the sink is still being called and checks whether the substring *marker*<>'" is present on the sink argument without any encoding. DOMsday authors sampled 40 cases that were flagged as vulnerable using this method and validated that all those cases were indeed vulnerable (i.e., true positives).

SWIPE adopts the same payload generation algorithm as DOMsday because its algorithm is open-source. For injection, SWIPE generates multiple confirmation URLs for each potential flow, one for each of the three injection methodologies. With respect to flow *uniqueness*, SWIPE deduplicates potential and confirmed flows using the same criterion adopted by all prior work, based on the domain and the vulnerability location [31]. The location is defined as the URL of the script without PFs and the line and column offset of the sink call within that script. For inline scripts, the URL and line offset are omitted.

D. DOM-XSS flow encoding

For a flow to be exploitable, the sink argument cannot be URL encoded, so the attacker can inject special characters like quotes to successfully inject arbitrary JavaScript code. Over the years, browsers have improved their built-in mechanisms that encode such characters if they are placed on the GET parameters or fragment value of the URL. In 2013, a third of the exploits generated by 25mFlows were successful against Internet Explorer because that browser did not have many of these built-in URL encoding mechanisms. In 2017, DOMsday injected its payloads on the fragment value, which was not encoded by default in the version of Chromium that DOMsday used. In 2020, TalkGen used a modified version of Firefox that disabled URL encoding. Nowadays, all modern browsers, including the Chromium browser that SWIPE uses, enforce URL encoding on the PFs, and Internet Explorer is discontinued. In Sec. IV, we discuss the impact of encoding in detection.

III. Methodology

We first present an overview of the end-to-end architecture of SWIPE, then describe its three novel components: 1) a fuzzer that simulates user interactions with a webpage to elicit new behaviors (Sec. III-B); 2) a DSE engine that synthesizes new URLs from the analysis of the constraints in the webpage to *discover* new parameters and fragments that can be of use

(Sec. III-C); and 3) a web archive that proxies all requests and responses to the target website to reduce the sources of randomness and improve reproducibility (Sec. III-D). SWIPE uses a version of Chromium from DOMsday [36] updated to the base Chromium version 126.0.6478.264 (released Feb. 7th, 2025). This browser has an instrumented V8 JavaScript engine that dynamically tracks taint of strings.

A. SWIPE *architecture overview*

Given a list of pages to be analyzed, we spawn multiple containers at the same time. Each container is responsible for analyzing a single page and running one of SWIPE's components, which we refer to as the analysis *condition*: Passive, Fuzzer or DSE. In Fig. 2, we present the end-to-end execution of the infrastructure against a single page inside a container. We start by launching a Chromium instance under a given condition and navigate to the target URL (①). The Chromium instance is configured to use our proxy that intercepts the navigation to the target URL (②). The proxy manages web archiving for reproducibility and optionally instruments the code for DSE.

The proxy loads the appropriate archived page and reads the content (e.g., HTML, JavaScript, CSS, etc.) corresponding to the requested path and query parameters. If no such web archive exists, then it contacts the live webpage and requests its contents (③, ④, ⑤). Otherwise, the live page is only requested if no similar request was archived previously. When running DSE (⑤c), regardless of being a fresh or an archived webpage, the proxy instruments the HTML and JavaScript files with our dynamic symbolic execution code using Jalangi2 [54, 55]. Then, Chromium loads the archived page content (⑥), which is instrumented in the DSE case (⑥c).

The remaining steps depend on the SWIPE condition that is running. Under the Passive condition, the browser simply loads the page and waits, executing JavaScript code until the time budget runs out (⑦a). The Fuzzer condition will execute UI fuzzing (see Sec. III-B) on the page guided by coverage information (⑦b). For the DSE condition, upon loading the page, the DSE analysis will execute and gather constraints to construct an SMT formula and attempt to solve it using Z3 (⑦c, see Sec. III-C).

For all conditions, once the time budget is exhausted, the final coverage is retrieved (⑧, ⑨). Passive and Fuzzer also return a list of flows that were found by the modified browser (⑩, ⑪), while DSE returns a list of derived URLs.

B. *Fuzzing user interactions*

Webpages use event handlers to define how user actions are dynamically handled. Fig. 3 shows a code snippet of real-world JavaScript from our crawl.[1] Lines 1–10 show a vulnerable function, which uses the GET parameter `q` (line 3), applies URL decoding to it (line 4), and creates a DOM element (line 6). Line 7 is the problematic assignment: attacker-controlled input from the URL (the value of `q`) is assigned to the `href` attribute

[1]To reduce the risk of exposing a vulnerability, we omit the URL and apply basic semantic-preserving code transformations to not be searchable

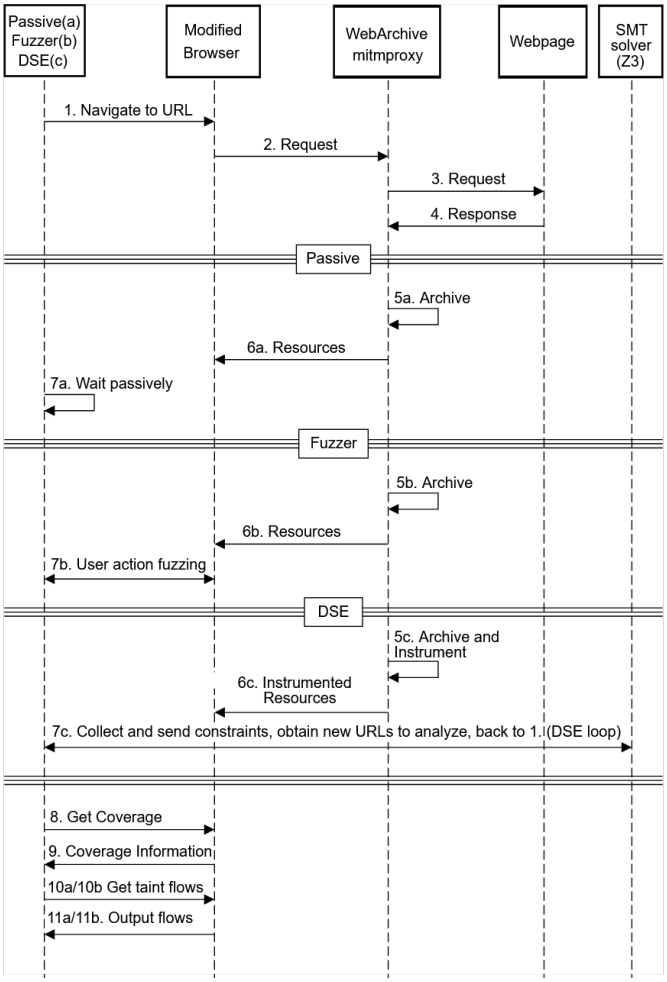


Fig. 2: Core proxy and browser interactions in an end-to-end SWIPE workflow for a single page. Specific component actions (of Fuzzer and DSE) occur within the browser context.

of the newly created DOM element. An attacker could thus pass, for example, `q=javascript:alert()` in the GET parameters. When `renderResult` is called, a malicious link is injected in the page, and when the link is clicked, attacker-supplied code will be executed. This example requires a user interaction, a click to trigger the event handler (lines 11–14), for the vulnerability to be explored and detected by dynamic taint analysis, as `renderResult` function (line 13) is **not** called as a result of page loading.

Next, we describe our fuzzer, which can automatically trigger event handlers on the page. Our fuzzer only reports vulnerabilities replicable by a real user interacting with the page. To achieve this, we do not simply force the execution of all event handlers available in the page, as this method could find vulnerabilities that cannot be triggered by a real user (e.g., by triggering event handlers on invisible DOM elements).

Our fuzzer supports a total of 55 event handlers. Given our target of fuzzing user interaction, we only support event handlers that depend on a user's actions, are supported by Chrome, and can be simulated through Chromium's DevTools

```
1  function renderResult(){
2      var search = location.search;
3      query = new RegExp(/[?&]q=([^&]*)/).exec(search);
4      query = decodeURIComponent(query[1]);
5      ...
6      var b = document.createElement("a");
7      b.href = query;
8      ...
9      element.appendChild(a)
10 }
11 document.addEventListener("click", (function(e) {
12     var a = e.target.closest(PAGE_LINK_SELECTOR);
13     a && renderResult()
14 }));
```

Fig. 3: Simplified vulnerable code found in the wild.

```
1  function getJsonFromUrl() {
2    var result = {};
3    var query = location.search.substr(1);
4    query.split("&").forEach(function (part) {
5      var item = part.split("=");
6      result[item[0]] = decodeURIComponent(item[1]); });
7    return result; }
8  ...
9  var paramsMap = {'custom1':'GP1', 'custom2':'GP2', ...};
10 function buildUrl(baseUrl) {
11   var urlParams = getJsonFromUrl();
12   for (var key in paramsMap) {
13     if (urlParams[key] != undefined) {
14       baseUrl += '&';
15       baseUrl += paramsMap[key] + "=" + urlParams[key];
16     }}
17   return baseUrl;
18 ...
19 var post = buildUrl(basePost);
20 document.write('<img src="' + post + '"/>');
```

Fig. 4: Vulnerable code requiring specific URL parameters

Protocol API. We provide a list of event handlers and their support from the fuzzer in our technical report [50].

1) Collect event handlers: Once a page is loaded, SWIPE's fuzzer iterates over all DOM elements and retrieves the associated event listeners via the `getEventListeners` function. In the example described in Fig. 3, the fuzzer learns that the target `document` has an `onclick` listener.

2) Generate high-level actions: Our fuzzer's units of input are user actions such as `ClickElement`, which we call high-level actions (HLA). The fuzzer manipulates HLAs, which can be parameterized, to interact with the webpage (e.g., `ClickElement` receives a X-path reference to the DOM element to be clicked).

Our fuzzer initializes a pool of actions by mapping supported event handlers into sets of triggering HLAs. For example, when the fuzzer finds an `onclick` event associated with DOM element `e`, it adds to the pool the following sequence of HLAs: `SequenceActions(MakeVisible(e),ClickElement(e))`

The `MakeVisible` action attempts to make a specified DOM element visible by scrolling down or up the page. Without it, the `ClickElement` action could try to move the mouse to coordinates that are not in the visible area of the page, deviating from what users can do.

To simulate interactions, the fuzzer uses Chrome DevTools Protocol (CDP), which allows programs to interact with the page. For the `ClickElement` action, the fuzzer uses CDP to first obtain the position of the element, then move the mouse to it and finally press and release the left button.

3) Coverage-guided fuzzer: Once a pool of actions is initialized, the fuzzer repeatedly executes the actions in the pool and removes, adds, or modifies actions from the pool based on the amount of code executed by previous actions, until its time budget is exhausted. After the initial round, the fuzzer combines actions sequences that achieved high coverage into composite actions. Each round, the fuzzer changes the pool of actions by only keeping those that executed the most code. It also exchanges inner actions between composite actions in the pool, a process known as cross-over [63], to obtain a diverse set of actions that trigger event handlers in different orders.

4) Replaying actions: To reproduce the HLAs generated by the fuzzer for later confirmation of flows, all HLAs that require randomness use a pseudo-random number generator (PRNG) initialized with the same seed. When confirming flows found by the fuzzer, we replay all fuzzing actions that were performed during analysis.

C. Synthesis of GET parameters and fragments

The execution of a page's JavaScript code can be affected by components of the page's URL, namely, the GET *parameters* and URL *fragment* (collectively named PFs), and certain execution paths may only be reachable if specific parameter keys and values, or fragment values are present.

A real-world example where the execution of a vulnerable page script depends on the presence of a specific key/value pair in its GET parameters is shown in Fig. 4. The script first builds a `urlParams` object (line 11) from attacker-controllable URL GET parameters keys and URL decoded values (lines 1–7). Next, selecting *only* the keys in `paramsMap` (line 9), it creates a new URL with GET parameters assigned to the attacker-controllable values from `urlParams` (lines 12–15). Finally, it writes the created URL to the DOM as the `src` attribute of an `img` (line 20). To find this vulnerability, we need to have a URL that has one of the allowed GET parameters, e.g., `'custom1'`. We employ Dynamic Symbolic Execution (DSE) to analyze and generate PFs that trigger these code paths.

1) DSE loop: An overview of the end-to-end execution of the DSE component is shown in Fig. 2. Given a URL of the form `domain.tld/path/?search#fragment`, Chromium will load the DSE-instrumented page content and treat the URL's `location.search` (parameters) and `location.hash` (fragment) as concolic values, which store both a symbolic and a concrete component. Upon loading, the DSE analysis begins to execute and gather the constraints on operations performed on these concolic values. Upon receiving a set of constraints from the analysis, DSE generates an SMT formula from these constraints and invokes an SMT solver, in our case Z3, to solve it. If the formula is satisfiable, the solver provides concrete values for the symbolic components. The DSE component then gets a URL with new query and fragment components that is added to a queue. The process is then repeated with a URL from the queue. We call this process a *DSE loop*.

```
1  function binary(op, lhs, rhs, res) {
2    if (isConcolic(lhs) || isConcolic(rhs)) {
3      res = handle(op, lhs, rhs) }}
4  function handle(op, lhs, rhs) {
5    if (op == "+") {
6      sendConstraints("str.++", lhs.smt(), rhs.smt())
7      return lhs.concrete + rhs.concrete }}
```

Fig. 5: DSE instrumentation for string concatenation.

The DSE instrumentation also records coverage information during execution: textual ranges of JavaScript script characters whose code was executed. Execution of the iterative DSE loop can be halted after a number of iterations, a given time budget, or a particular coverage threshold is reached.

2) DSE instrumentation and concrete models: In Fig. 5, we show an example of DSE instrumentation for binary operations, which intercepts the string concatenations in Fig. 4 (lines 15 and 20), generating constraints that link the attacker-controlled `search` parameter to the final URL written to the DOM sink.

Line 1 shows the instrumented function `binary`, which takes the operation (`op`), left-hand side (`lhs`), right-hand side (`rhs`), and a placeholder for the result (`res`). Line 2 checks if either operand is concolic (i.e., derived from the URL parameters or fragment), and if so, then line 3 calls the `handle` function to process the symbolic operation. When evaluating an operation involving a concolic value, the instrumentation calls a handler (like `handle` in Fig. 5), which generates SMT constraints representing the operation's effect on the symbolic value. The `handle` function (lines 4–7) generates SMT constraints based on the operation. For string concatenation (`op == "+"` on line 5), it sends a `str.++` constraint to the solver via `sendConstraints` (line 6), using the SMT representations of the operands (`lhs.smt()`, `rhs.smt()`).

It then evaluates the operation using a *concrete model* that replicates the exact semantics of the original JavaScript operation using the concrete parts of the concolic values (e.g., `lhs.concrete + rhs.concrete` on line 7). This ensures that the page's JavaScript execution proceeds normally with a valid concrete value, even when dealing with concolic inputs. The instrumentation preserves the original semantics for operations involving only concrete (non-concolic) values.

Our concrete models are carefully implemented JavaScript functions that mimic the behavior of native JavaScript operations, especially for strings, as our initial symbolic inputs (`location.search` and `location.hash`) are strings. Ensuring the models are correct is challenging due to JavaScript's complex semantics and edge cases. For example, the native string operations `charCodeAt` and `codePointAt` have similar semantics but differ subtly in their return values for out-of-bounds indices: `charCodeAt` returns `NaN`, while `codePointAt` returns `undefined`. Our concrete models must replicate these specific behaviors precisely, as page code might rely on them (e.g., a branch checking specifically for `NaN` would behave differently if `undefined` were returned instead).

Our instrumentation and concrete models should not alter the observable behavior of the JavaScript code compared to its native execution, except for the collection of constraints. This property is commonly referred to as *semantic transparency*. We strive for semantic transparency because altering the execution flow could lead the DSE to explore irrelevant paths or miss vulnerabilities present in the original code. We validated the *semantic transparency* of our models using a comprehensive suite of 357 unit tests covering numerous edge cases and standard behaviors for each modeled JavaScript operation, according to the ECMAScript specification [8].

$1: [(\texttt{not}\ (\geq\ (\texttt{indexOf}\ a\ b\ 0)\ 0))]_\kappa \hookrightarrow (=\ (\texttt{indexOf}\ [a]_\kappa\ [b]_\kappa\ 0)\ -1)$

$2: [(\texttt{not}\ (=\ -1\ (\texttt{indexOf}\ a\ b\ 0)))]_\kappa \hookrightarrow (\geq\ (\texttt{indexOf}\ [a]_\kappa\ [b]_\kappa\ 0)\ 0)$

Fig. 6: `String.indexOf` transformers used by DSE.

3) DSE string constraint solving: SMT constraints associated with common JavaScript string operations such as substring extraction, substring membership, and substring replacement fall into the category of *extended string terms*, whose solving is undecidable in general [49]. Different semantically equivalent encodings of a set of constraints can vary in solver performance; for example, modeling a JavaScript string operation via SMT array operations can be inefficient [6]. We develop a rewriting algorithm that takes an SMT formula and first applies a set of built-in Z3 simplifiers [6], followed by a set of custom *transformers* we define. These transformers cover common patterns of string operations in web scripts.

The general form of the transformer is $[s]_\kappa \hookrightarrow [s']_{\kappa'}$; the transformer takes an SMT statement AST s and context κ and transforms it to a statement s' and context κ'. The transformer can be applied recursively within the structure of s'. The context κ is a set of new string constant declarations that are introduced by the transformer. In Fig. 6, we show an example transformer for rewrite constraints that involve the `indexOf` operation. We re-encode comparisons of string `indexOf` operations where the solver is being asked if a substring is *not* at a position ≥ 0 in the string. This asks Z3 to check if the position is any negative integer. However, there is only a single negative integer result of `indexOf` permitted by the JavaScript semantics, that is -1. We thus reduce the search space by equivalently asking the solver if the substring index in the string is exactly -1.

To handle the cases where rewriting worsens solving time, we implemented a portfolio solving strategy, running two solvers in parallel: one Z3 instance solves rewritten constraints; another solves the original constraints. The portfolio solver then returns the result of whichever solver returns first. This strategy enables us to take advantage of the cases where rewriting improves solving time, while still being able to fall back to the original SMT formula if rewriting increases the run time.

D. Web archive

The dynamic nature of the web makes it challenging to fairly compare web analysis approaches. For example, a page found in our crawl chooses which resources to load based on a random number (Fig. 7). A key feature of SWIPE that enables repeatable page visits, and hence fair cross-condition

```
var vlst = ['vid_01', 'vid_02','vid_03', 'vid_04', ...];
var vid = vlst[Math.floor(Math.random() * vlst.length)];
document.write('<video poster=
                "http://vulnerable/' + vid + '.jpg">');
```

Fig. 7: Randomized page behavior observed in a real page

comparison, is website archiving. Though benefits of using web archiving as a vehicle for repeatable analysis have been demonstrated by prior work [9, 19], adapting web archiving for DOM-XSS required addressing specific challenges.

DOM-XSS often involves complex client-side JavaScript that dynamically fetches resources or modifies requests based on user interactions or URL fragments, which are not always captured adequately by standard archiving crawls. Ensuring that the archive replay mechanism correctly handles these dynamic requests, including those involving symbolic parameters generated by DSE, posed significant engineering challenges. Our contribution lies in integrating the archive with dynamic analysis techniques (Fuzzer and DSE) and implementing robust replay mechanisms, including online archive expansion and similarity matching, to handle the dynamic nature of modern web applications susceptible to DOM-XSS. A novel aspect of our approach is automatic expansion of a web archive to reduce missed resources, while maintaining repeatability.

1) Archive construction: If a web archive does not exist for some target page, one is created for it the first time the page is analyzed. When the page is first visited, we continuously store all the page's HTML, JavaScript, and CSS from server responses intercepted by our proxy. We additionally store all the requests initiated by the page that resulted in those responses. This information is stored in a compressed format called a *web archive*; we use the WARC format [4]. Our system handles HTTP redirections robustly: if a request for URL_1 during archiving results in a redirection to URL_2, we store the response for URL_2 in the archive and maintain a redirect mapping from URL_1 to URL_2.

2) Archive replay: If a web archive exists, rather than visiting the page again, we instead load it from our web archive. Requests made by a page are intercepted, and we load the corresponding response entries from our web archive. To handle redirects during replay, we reference the redirect mapping created during archiving. New requests generated by the page (e.g., through scripts) may not be present in the archive. Often, the differences in the request come from differences in dynamic elements such as timestamps in URL query parameters. For these cases, we calculate the URL similarity based on the methodology proposed by Goel et al. [9] to find the nearest URL to the one requested. If no similar URL can be found, we execute an online archive expansion phase where a request to the target URL is issued to the live web server, and the response is saved to the archive. We can then repeat visits for each condition with the extended archive. These strategies ensure that dynamically generated yet semantically similar requests can often be served from the archive, maximizing repeatability across different analysis conditions.

IV. Evaluation

In this section, we evaluate the effectiveness of our approach and answer the following research questions:

RQ1: Can Fuzzer generate user interactions in real-world pages that lead to the discovery of new DOM-XSS vulnerabilities?

RQ2: Can DSE uncover PFs in real-world pages, and how do they impact DOM-XSS detection?

RQ3: How does SWIPE compare to other end-to-end DOM-XSS detection tools?

RQ4: How do SWIPE's analysis results compare to what prior work reported, and how does the continuous evolution of the web affect the validity and consistency of such comparisons?

A. Experimental setup

The first time each page is analyzed by a SWIPE component, a web archive for that page is created (Sec. III-D). We ensure that each page is analyzed by one condition at a time to prevent race conditions in the archive creation. All SWIPE components analyzing the same page use the same archive, but we disable the web archive during confirmation to ensure that any discovered vulnerabilities stem from real page behavior.

Dataset collection. We pre-crawled the top 30,000 domains from the Tranco list [30] and extracted a maximum of 5 subpages from each page. We only include a subpage if it contains GET parameters and has the same domain as one of the top 30,000 pages. This results in a dataset of 44,480 URLs after removing timed-out pages: 13,396 top-level Tranco pages and 31,084 subpages with GET parameters. We call this dataset our *Core dataset*. All SWIPE components, together with our replication of TalkGen, found a total of 194 pages with DOM-XSS vulnerabilities in our Core dataset. We call the full URL of these pages our *Vulnerable dataset*. In some of our experiments, we will strip the GET parameters and fragments from the URLs in these datasets. We call these stripped datasets *Core-noPFs* and *Vulnerable-noPFs*, respectively.

Dataset augmentation. We use SWIPE's DSE component to synthesize PFs for a set of URLs and call this process *dataset augmentation*. The difference of results obtained from a dataset and the results obtained from its augmentation reveal the impact of DSE and answer **RQ2**.

We performed two runs of DSE with different time budgets and starting datasets, always stripping the PFs from the initial URLs before passing them to DSE. Even though DSE may perform better if it was given the initial GET parameters, we believe stripping the PFs from the target URLs prior to DSE augmentation leads to cleaner evaluation results because DSE is also evaluated for PFs' rediscovery capabilities on the same dataset. The time budget refers to how much time DSE has to analyze a single URL and come up with as many derivative URLs as possible. First, we run DSE with a 1-hour timeout per page on a random subset of URLs from our Core-noPFs dataset. We do this to understand whether DSE with a reasonable timeout can help us find new confirmed flows on pages in the wild. We allowed this process to run for 100h in our infrastructure and managed to go through 13,555 URLs

of the Core-noPFs dataset, generating 56,152 new URLs (that we call *Core-DSE1 dataset*). We also run DSE with a 24-hour timeout on the Vulnerable-noPFs dataset. The objective here is to figure out if DSE rediscovers the relevant PFs that contributed to triggering vulnerabilities. Running this process over the 194 URLs generated 5,417 new URLs (that we call *Vulnerable-DSE24 dataset*). The original PFs in URLs of the Core dataset were found in the *dataset collection* pre-crawl.

Conditions and their results summary. We evaluated the following seven main conditions. We discuss the *rationale* behind each condition and present detailed results in the upcoming sections. **(1) Passive**: This is the baseline replicating passive navigation from DOMsday [36] with an upgraded browser (Chromium 126) on the Core dataset. It uncovered 72 confirmed flows on pages hosted in 64 unique domains. **(2) Fuzzer**: Our fuzzer that simulates user interactions on the Core dataset. It uncovered 83 confirmed flows in 73 domains. **(3) Fuzzer-noPFs**: Our fuzzer on the Vulnerable-noPFs dataset. It found 32 confirmed flows in 28 domains. **(4) Fuzzer-DSE24**: We run Fuzzer on the DSE augmented version of the Vulnerable-noPFs dataset (Vulnerable-DSE24 dataset). This found 42 confirmed flows in 29 domains. **(5) Fuzzer-DSE1**: We run Fuzzer on the DSE augmented version of the Core-noPFs dataset (Core-DSE1 dataset). This found 15 confirmed flows in 9 domains. Note that both Fuzzer-DSE1 and Fuzzer-DSE24 crawls only apply fuzzing to URLs not already in the original datasets (i.e., before augmentation). This avoids duplicate work, as the Fuzzer condition already applies fuzzing to all URLs in the Core dataset, and also explains why the number of confirmed flows resulting from those DSE crawls are both smaller than 83. **(6) FoxHound-ENC**: We run FoxHound [29] (TalkGen's taint-enabled browser) on the Core dataset, but we run FoxHound v126.0, the latest release at the time of the crawl, as an end-to-end tool, without using our web archive. In this version of FoxHound, URL encoding was re-enabled for fair comparison with our browser. This resulted in 68 confirmed flows in 58 domains. **(7) FoxHound-2025**: We run FoxHound with no modifications. While it reported 347 confirmed flows in 250 unique domains, we manually analyzed a sample of these and found no more exploitable flows (in modern browsers) than those already found with FoxHound-ENC. We also launched FoxHound-ENC, Passive-live (i.e., Passive without the web archive) and Fuzzer-live against the Vulnerable dataset and measured how long each condition took from opening the browser to rediscovering a previously found confirmed flow. Results are summarized in Fig. 12, where the X axis is time. For all conditions, the performance in detecting DOM-XSS rises quickly at the start and plateaus a few minutes later. Particularly, this happens earlier with FoxHound-ENC, as expected considering the known coverage tracking overhead [18] in our Chromium, a necessary feature for our fuzzer component. A summary with all the performed experiments can be found in our companion tech report [50].

Run time. The analysis of each page and condition is run on an isolated Docker container restricted to 6GB RAM and 6 cores. For all conditions but DSE, we enforce a 3-minute timeout per page, plus 1 minute to gracefully exit before termination.

Flow Confirmation. SWIPE uses the same confirmation methodology as DOMsday. To validate this methodology, we reviewed 10 confirmed flows and successfully exploited all 10 by creating a payload that spawned an alert window.

B. RQ1: Importance of user interactions

We evaluate whether user interactions generated by our fuzzer can discover new vulnerabilities by comparing the results of Passive and Fuzzer on the Core dataset. Fig. 8a shows the number of unique potential flows found by Passive and Fuzzer. Fuzzer increased the number of unique potential flows compared to Passive from 2023 to 2449, a 21% increase. Fig. 8b shows the number of confirmed flows found by the same conditions, where we can see a 15% improvement from 72 confirmed flows found by Passive to 83 found by Fuzzer.

Given the randomness associated with Fuzzer, and to a lesser extent with Passive, we run Fuzzer and Passive up to 5 times on the pages that contained the 21 flows uniquely identified by one of the conditions. From the 16 flows uniquely identified by Fuzzer, 15 were found when we simulated user actions but never when we passively stayed on the page. In the remaining case, Fuzzer and Passive actually found the same vulnerability, but the script that contained the sink call had slightly different content; thus, the script location differed across conditions. This can happen, for example, when random or time-based requests are made to the page, which are not handled by our web archive because that component is disabled during confirmation. The other 4 of the 5 confirmed flows that Passive uniquely found were also found by Fuzzer in a second retry.

We also evaluate the benefit of triggering actions and combining actions into composite actions, in terms of executed code and confirmed flows detected. For that, we analyzed all 194 vulnerable pages with a version of Fuzzer that simply keeps in the pool the units of HLA that achieved the most coverage every round, without combining them into composite actions. We call it *simpleFuzzer*. This version misses 7 of the 83 confirmed flows that are found by the original Fuzzer. Fig. 8c shows that more bytes of JavaScript are executed: from 1,298,737 bytes on average by Passive, to 1,364,087 by simpleFuzzer, to 1,468,703 bytes by the original version of the Fuzzer, a 13% increase from Passive, and an 8% increase from simpleFuzzer. The increase is due to the executed event handler code and the additional resources the fuzzer loaded by executing that code. Fuzzer is critical to executing a significant number of unique event handlers: on the 194 vulnerable pages, Passive executes a total of 291 supported event handlers and simpleFuzzer executes 1,762, while Fuzzer executes 2,920 (detailed breakdown in Fig. 15, App. A-A). Passive also executes some event handlers because they are triggered programmatically by the page code, without user interactions.

Finally, we compare the performance of Fuzzer with CrawlJax [38]. CrawlJax systematically explores JavaScript-driven web applications through automated interaction but lacks

built-in DOM-XSS vulnerability detection. To surpass that limitation, we integrated CrawlJax with our modified browser and compared its results in our Vulnerable dataset with those reported by Fuzzer under the same run-time conditions. Results are summarized in Table II. CrawlJax found 47 confirmed flows in 40 domains; 19 of these flows and 1 domain not identified by SWIPE-Fuzzer. SWIPE-Fuzzer found 55 confirmed flows in 34 domains that CrawlJax did not. In one CrawlJax-discovered flow, SWIPE-Fuzzer resized the browser window (to trigger an `onresize` handler), which hid a DOM element required for vulnerability activation. In the remaining 18 cases, SWIPE-Fuzzer identified the same vulnerabilities but with different hashes due to minor variations in script names or content, which shifted sink locations. We only encountered one such case during the Fuzzer vs. Passive analysis. We suspect the slight changes in page content originated from the different way CrawlJax interacts with the browser, compared to any SWIPE condition. We launched a version of the Fuzzer that passes identical browser flags and browser window size as CrawlJax. In 5 out of 18 cases, we found a flow with the same hash as CrawlJax did, and when we reverted to the original browser flags and window size, we found the same flow hash as SWIPE-Fuzzer did in our original crawl.

Component/Tool	Conf Flows	Domains
SWIPE-Fuzzer	83 (55)	73 (34)
CrawlJax+Taint-tracking Chromium	47 (19)	40 (1)

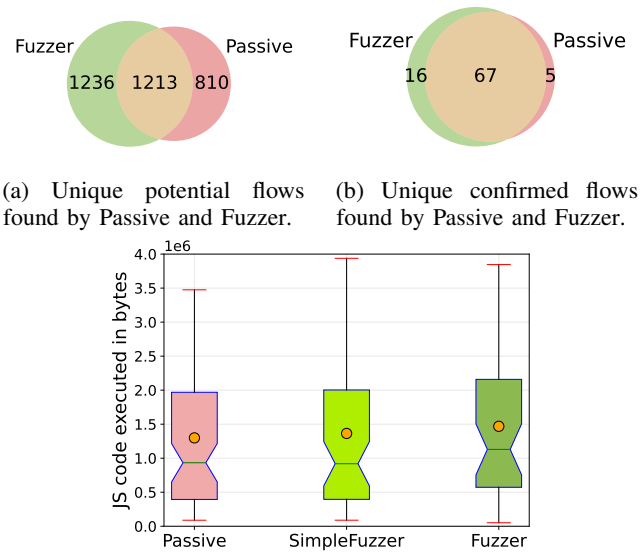
TABLE II: Number of confirmed flows and vulnerable domains detected by SWIPE-Fuzzer and CrawlJax on the Vulnerable dataset. Numbers in ( ) indicate how many flows and domains are unique to each tool. CrawlJax+Taint-tracking Chromium refers to CrawlJax using our browser to detect flows.

Result 1: SWIPE's fuzzer was responsible for a 21% increase in potential flows and a 15% increase in confirmed flows over Passive. CrawlJax found 19 of confirmed flows not identified by SWIPE-Fuzzer, while SWIPE-Fuzzer found 55 confirmed flows that CrawlJax did not. These results and the number of event handlers that are not yet executed indicate that simulating user interactions is a promising avenue for increasing DOM-XSS detection.

C. RQ2: Synthesis and impact of PFs

DSE aims to synthesize URL PFs (Sec. III-C). To assess its effectiveness, we refine RQ2 in the following sub-questions:
RQ2a: How effective is DSE at synthesizing PFs that can discover DOM-XSS vulnerabilities?
RQ2b: Does analyzing DSE-generated URLs increase DOM-XSS flows found?
RQ2c: How does DSE compare in effectiveness to off-the-shelf GET parameter discovery tools?

1) RQ2a: Effectiveness of DSE at synthesizing vulnerability-triggering PFs: The trigger of a vulnerability may depend on the presence of specific keys/values in the parameters/fragments. The Core dataset that RQ1 uses already contains PFs found



(a) Unique potential flows found by Passive and Fuzzer.

(b) Unique confirmed flows found by Passive and Fuzzer.

(c) JavaScript bytes of code executed for Passive, Fuzzer and Fuzzer without action combinations in the Vulnerable dataset.

Fig. 8: Comparison of flows and coverage across different conditions (RQ1).

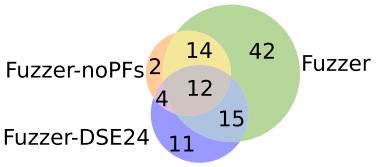


Fig. 9: Comparison of confirmed flows found across Fuzzer, Fuzzer-noPFs and Fuzzer-DSE24 (RQ2a & RQ2b).

in the wild. One question we address is whether DSE can synthesize the necessary PFs if these are not known and are absent in the target URL. With this RQ, we evaluate DSE's ability to both rediscover known vulnerabilities and to uncover new ones through parameter synthesis.

First, we use the Vulnerable-noPFs dataset, whose URLs have no PFs, to assess whether DSE can automatically synthesize the necessary PFs to rediscover known vulnerabilities. Fig. 9 shows the number of confirmed flows obtained with Fuzzer-noPFs on the Vulnerable-noPFs dataset, Fuzzer-DSE24 on the Vulnerable-DSE24 dataset (Vulnerable-noPFs dataset augmented with synthesized PFs), and of Fuzzer on the Vulnerable dataset (the original 194 URLs of the Core dataset). There is a set of 57 $(42 + 15)$ confirmed flows that Fuzzer finds only when the PFs are included on the target URLs. The other 26 (14+12) are identified by the Fuzzer-noPFs, which imply that the PFs are not needed. When we strip these PFs and apply DSE to synthesize new ones, Fuzzer-DSE24 is able to rediscover 15 out of these 57 flows (26%). Notably, Fuzzer-DSE24 also discovered 11 unique confirmed flows on these pages. To verify that they require PFs synthesis, we launched

Fuzzer and Passive three times against these flows, and 10 out of the 11 were not discovered without DSE.

Next, we examine whether DSE can discover vulnerabilities in pages collected from the wild after we strip the PFs. For that, we used the Core-DSE1 dataset of 56,152 URLs with synthesized PFs (see Section IV-A). After running Fuzzer on these URLs, we find 15 confirmed flows. A manual analysis of these 15 flows reveals that 10 of them are previously undiscovered vulnerabilities (in 7 pages hosted on 5 unique domains, 4 of these being new vulnerable domains). The remaining 5 confirmed flows were previously discovered by the Fuzzer when it ran against the Core dataset. To synthesize each URL in the Core-DSE1 dataset that led to vulnerabilities, DSE required a time ranging from 7 to 3410 seconds, with an average of 1048 seconds (~17 minutes). Details regarding DSE scalability can be found in Section A-B, which focus on our portfolio solving strategy performance evaluation.

Result 2a: DSE is effective at synthesizing vulnerability-triggering PFs, both for rediscovering known vulnerabilities and for uncovering new ones. It successfully synthesizes PFs to rediscover 26% of confirmed flows that require specific parameters, while also generating entirely new parameter combinations that reveal 10 new vulnerabilities on known vulnerable pages and 10 previously-undiscovered vulnerabilities in other pages of the Core dataset.

2) RQ2b: Impact of DSE-synthesized PFs on uncovering DOM-XSS flows: We examine the combined impact of Fuzzer augmented with DSE to understand how synthesized parameters enhance vulnerability detection.

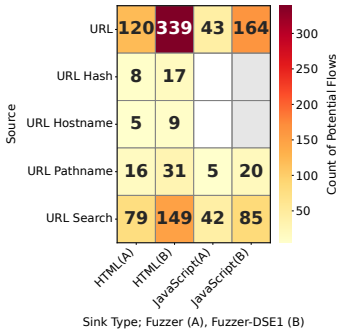


Fig. 10: Heat map of potential flows for both Fuzzer (A) and Fuzzer-DSE1 (B) by flow source and sink (RQ2b).

We compare the results of Fuzzer (condition A) with Fuzzer-DSE1 (condition B) on 5631 pages from the Core dataset analyzed by both conditions. Note that Fuzzer-DSE1 might be using different PFs from those in the Core dataset. Within this set, Fuzzer finds 325 potential flows, while Fuzzer-DSE1 finds 823 potential flows. In Fig. 10, we report the counts of potential flows found, striated by DOM-XSS sources and sinks.

The heat map reveals a consistent pattern: the use of DSE leads to additional flows in all source-sink pairs. The difference is particularly large for flows from URL (`location.href`) to JavaScript sinks (43 vs 164; 281% increase), URL to HTML sinks (120 vs 339; 183% increase), URL search (parameters) source to JavaScript sinks (42 vs 85; 102% increase), and URL search to HTML sinks (79 vs 149; 89% increase). Each of these source-sink flows are critical vectors for DOM-XSS.

Finally, reviewing the confirmed flows with a 24-hour DSE timeout (Fig. 9), Fuzzer-noPFs finds 32 confirmed flows, while Fuzzer-DSE24 finds an additional 26 confirmed flows requiring PFs that were synthesized by DSE.

Result 2b: DSE-uncovered PFs significantly increases detected potential flows compared with Fuzzer, particularly from URL sources to JavaScript and HTML sinks, with increases of 281% and 183% respectively. Using the fuzzer with DSE leads to 26 additional confirmed flows. This indicates that synthesizing PFs is crucial for enhancing DOM-XSS detection capabilities.

3) RQ2c: Comparison with off-the-shelf GET parameter discovery tools: We compare DSE against two off-the-shelf fuzzers aimed at uncovering GET parameters: ffuf [20] and wfuzz [3]; see Sec. VI for an overview of their methodology. We focus on the Core-DSE1 dataset, which includes 6,549 unique, DSE-generated GET parameters keys. We download the default *wordlists* used by ffuf and wfuzz, containing 50,275 unique entries total, which captures all unique GET parameters that these tools would try. Then, we compute the intersection between the wordlists and DSE-found GET parameters, and find that they overlap in just 287 entries, indicating that 95.6% of the unique GET parameter keys synthesized by DSE could not be found by ffuf nor wfuzz.

We also compared our DSE with Wapiti [59]. Wapiti is a web scanner that can discover GET parameters mostly by parsing forms or links present on the page. We evaluate its ability to generate relevant GET parameters by fuzzing URLs generated by Wapiti and comparing the results with our DSE condition on the Vulnerable-noPFs dataset.

We gave Wapiti a time budget of 24 hours per page, to fairly compare it with our DSE condition. In both cases, we launched SWIPE-Fuzzer on the generated URLs to try to find vulnerabilities. The results are summarized in Table III. Wapiti was able to discover at least 1 GET parameter in 9 of the 194 pages, while DSE discovered at least 1 GET parameter in 101 of the 194 pages. Wapiti found GET parameters that DSE did not, so even though DSE is not a complete replacement for Wapiti, none of the vulnerabilities found by Wapiti + SWIPE-Fuzzer were new, having been previously discovered by SWIPE-Fuzzer alone. We also note that every GET parameter discovered by symbolic execution should cause different behavior in JavaScript (execution of a different program path) and consequently increase code coverage.

Result 2c: DSE can uncover GET parameters that off-the-shelf web fuzzers do not find. ffuf and wfuzz fail to find 95.6% of the unique GET parameters found by DSE on 13,555 pages. GET parameters found by DSE enable more confirmed flows to be discovered than those found by Wapiti. However, they find different parameters and are complimentary to each other.

Component/Tool	Conf Flows	Domains
SWIPE-Fuzzer (baseline)	83	73
SWIPE-DSE-24 + SWIPE-Fuzzer	98 (15)	78 (5)
Wapiti + SWIPE-Fuzzer	87 (4)	73 (0)

TABLE III: Number of confirmed flows and vulnerable domains detected by SWIPE-DSE-24 and Wapiti. Numbers in parentheses indicate the number of flows unique to each tool.

D. RQ3: Comparison with other DOM-XSS detection tools

In this section, we compare SWIPE with other DOM-XSS detection tools ZAP [45] and FoxHound. While ZAP is a web application security scanner that performs automated crawling and active scanning for vulnerabilities such as DOM-XSS, FoxHound is more similar to Passive, leveraging dynamic taint analysis in a modified browser with passive navigation.

SWIPE versus ZAP. ZAP was given a 3-minute timeout, similarly to our Fuzzer condition. When running ZAP, we included the spider AJAX and DOM-XSS active scan components and measured XSS alerts reported by the tool. As reported in Tab. IV, ZAP was only able to find vulnerabilities in 2 of these 194 vulnerable pages. Note that ZAP only reports XSS vulnerabilities it can successfully exploit by causing an alert window to open. ZAP often injects payloads in the hash to avoid being blocked by WAFs, but not all vulnerabilities are exploitable via the hash. Still, the results we have obtained are in line with what DOMsday previously reported regarding Burp, another web application scanner. We believe that the biggest factor in the increased performance when using our tool compared to ZAP stems from the power and flexibility of dynamic taint analysis, finding flows even when attacker input is transformed throughout JavaScript execution.

Component/Tool	#Vuln. pages	(%)
SWIPE + FoxHound-ENC	**194**	**100%**
SWIPE-DSE + SWIPE-Fuzzer	146	75.26%
SWIPE-Passive	127	65.46%
FoxHound-ENC	120	61.86%
ZAP	2	1.03%

TABLE IV: Number of pages from the Vulnerable dataset were deemed vulnerable by ZAP, SWIPE and FoxHound-ENC.

Passive versus FoxHound. To identify subtle differences between Chromium-based SWIPE and FireFox-based FoxHound, we run a taint-enabled FireFox browser from prior work [5] (FoxHound-ENC) and compare it with Passive against our Core dataset. The results are shown in the first two columns of Tab. V. We note that although the original work TalkGen [5] used FoxHound with URL encoding disabled, we ran FoxHound with URL encoding re-enabled to ensure a fair comparison with SWIPE which also has URL encoding enabled (and it is the current standard for modern browsers). Although Passive found fewer potential flows (2,023 compared with 3,408 found by FoxHound-ENC), Passive found a slightly higher number of confirmed flows (72 vs 68) hosted in a higher number of unique vulnerable domains (64 vs 58).

We manually investigated two potential flows that only FoxHound-ENC attempts to exploit, and we found that both are false positives. FoxHound-ENC conservatively treats the entire URL as a single source, even though the protocol and host portions of the URL cannot be reasonably controlled by an attacker. SWIPE instead treats each part of the URL (protocol, host, path parameters, GET parameters and fragment) as separate sources. As a result, SWIPE can easily discard harmless sources and consequently not try to exploit flows from the protocol or host portions of the URL to sinks. In one of those examples, the script takes the value of `location.href`, extracts the host part of that URL, and includes it in a value being assigned to the `innerHTML` attribute of an element of the DOM. Changing the host part of the URL would make the browser navigate to an entirely different website, and therefore, this flow is not exploitable. The other example is a potential flow from the protocol part of the URL (i.e., just the *https* part of `location.href`) that was reported as a potential flow by both tools, but only FoxHound-ENC tried to exploit it.

We directly used the potential flow reporting mechanism provided by FoxHound. FoxHound additionally provides an operation tree for each flow representing the operations that were performed on the source until it reached the sink, which can be used to infer which specific parts of the URL flow to the sink. If we were to write a post-processing script to remove the above-mentioned flows, we suspect we would obtain a number of potential flows similar to that reported by Passive.

Passive and FoxHound-ENC found 57 confirmed flows in common. Passive found 15 confirmed flows that FoxHound-ENC did not. FoxHound-ENC found 11 confirmed flows that Passive did not find. To analyze this difference, we manually sampled 5 confirmed flows that only Passive finds. Undertainting issues in FoxHound were the cause of 4 of these cases. The remaining case was due to overtainting: some bytes in the final sink argument were being wrongly reported by FoxHound-ENC as originating from the URL. This caused the injection algorithm to fail to locate an appropriate position for the payload to be injected in the iframe URL. We also manually sampled 5 confirmed flows that only FoxHound-ENC finds. Undertainting issues in the underlying Chromium of Passive caused all 5 missing cases. This demonstrates that catching all potential flows is difficult in reality, as modifying browser implementations is a huge undertaking.

Earlier discussion on the difference between SWIPE and FoxHound in treating URL sources also explains the difference between the total flows reported by Passive versus FoxHound-ENC (15.6M vs 3.8M). When all bytes of the URL flow to a sink, Passive reports multiple flows, one for each part of the URL (protocol, host, path parameters, GET parameters and fragment), whereas FoxHound-ENC reports a single flow. Though FoxHound supports more than DOM-XSS-related sources and sinks, the numbers reported in Tab. V for Passive and FoxHound-ENC do not include such flows.

Metric	Passive	FoxHound ENC (2025)	FoxHound No ENC (2021) [5]	DOMsday [36]	25m Flows [31]
Date	04/2025	04/2025	09/2020	08/2017	11/2013
Domains	30,000	30,000	100,000	10,000	5,000
Sub-pages	5	5	10	5	all depth 1
Web pages	44,480	44,480	390,092	44,722	504,275
Flows	15,647,717	3,826,017	20,912,107	4,140,873	24,474,873
Flows/1k pages	351,792	86,017	53,608	92,591	48,534
Potential	2,023	3,408	15,710	5,217	?
Pot./1k pages	45.48	76.62	40.27	116.65	?
Confirmed	72	68	7,199	3,219	8,163
Conf./1k pages	1.62	1.53	18.45	71.98	16.19
Vuln. domains	64	58	711	364	480

TABLE V: Crawling comparison between Passive and results reported by TalkGen [5] (FoxHound-2021, encoding disabled), DOMsday [36], 25mFlows [31] and FoxHound-ENC (encoding enabled), including number of flows, which include all source sink pairs considered by DOMsday, potential flows (Pot.), which only include URL sources to JavaScript or HTML sinks, and confirmed flows (Conf.).

E. RQ4: DOM-XSS detection over the years

In this section, we compare results from major efforts in identifying DOM-XSS in the past decade: SWIPE (2025), TalkGen [5] (2021), DOMsday [36] (2017), and 25mFlows [31] (2013). We discuss how the evolution of the web impacts the results and the challenges in replicating prior results and directly comparing the numbers. Tab. V shows a summary of the results. The first column includes results from running Passive (our DOMsday replication using Chromium 126); the second column is from running FoxHound-ENC (our TalkGen replication with URL encoding re-enabled); the next three columns are results taken directly from the published papers of TalkGen, DOMsday, and 25mFlows, respectively.

Recent results vs. previously reported results. We now compare our recent results with those of prior work, which are shown in the last three columns in Tab. V. The most important metric is the number of confirmed flows per 1,000 pages (Conf./1k pages), as the number of analyzed pages is different for each work. On a first look, Tab. V shows an apparent decline of DOM-XSS vulnerabilities. However, a deeper investigation reveals that several factors, such as dataset and methodology, may contribute to the dwindling number of confirmed unique DOM-XSS vulnerabilities.

First, modern browsers encode any special characters in URLs. The first two columns in Tab. V are results with URL encoding enabled. When we re-run the confirmation stage of FoxHound but with URL encoding disabled, we quintuple the number of confirmed flows, from 68 to 347 (7.8 confirmed flows/1k pages); however, these extra confirmed flows would not be exploitable in any modern browser. In fact, the three prior works did not consider URL encoding in their crawls: FoxHound-2021 did not consider any form of URL encoding, DOMsday did not consider URL encoding of the hash (and their payloads were always injected in the hash), and 25m flows used Internet Explorer to specifically confirm flows that had the *search* source, which did not perform URL encoding. Given that all modern browsers consider URL encoding, the lower numbers in the first two columns reflect real users' experiences.

Second, the definition of "unique" flows and vulnerabilities may differ across these studies. All prior works report that they use the same deduplication method as in the 25mflows work; however, the deduplication method was not rigorously defined in the paper [31]. The paper stated that the location of the sink call is one of the properties that should be used to deduplicate confirmed flows. One interpretation of the *sink location* is the full URL of the vulnerable script and the line and the column within the script where the sink call is. However, including the GET parameter in the URL of the vulnerable script for deduplication inflates the number of confirmed flows from 347 to 612 (13.76/1k URLs). Different GET parameters can and often do result in the loading of the same page; thus, we argue that these 612 flows are unlikely to be unique. Interestingly, 62% of all confirmed flows found by DOMsday were on a single domain, and the top 10 most vulnerable domains were responsible for 84% of the 3,219 confirmed flows [36], raising the question of whether using a deduplication method that does not include GET parameters would reduce these numbers.

DOMsday reported that the vast majority of vulnerabilities (82%) were found in web advertisement or analytics pages. In-browser mechanisms for content filtering (e.g., advertisement blocking) have evolved since prior work was published. Both Chromium [11, 12, 14, 15, 16, 17] and Firefox [39, 40, 41, 42, 43] block more advertisements and malware now than they used to when DOMsday and FoxHound ran their crawls. For DOMsday, 44.3% of all frames loaded are flagged as advertisements, while in Passive's case, which uses a very recent version of Chromium, only 26.26% of frames were flagged as advertisements. Fig. 11 shows a categorization of vulnerable top-level websites, frames and scripts that were found in our crawl. We used the IAB taxonomy [21] for categorization. The service Blue Coat K9 that DOMsday used is discontinued. While the distribution of vulnerable categories of top-level navigations is similar to what DOMsday reported, advertisement scripts are no longer the most vulnerable script category, ranked second after the "Technology & Computing" category. We hypothesize that this difference stems from browsers' improved ability to block malicious advertisements. For instance, DOMsday's Chromium did not filter intrusive ads, whereas our Chromium version does [12, 14, 16]. We kept ad filtering for the same reason as URL encoding: experiments should reflect modern browser behavior.

Finally, analysis of different datasets may also contribute to the difference in confirmed flows across different studies. We crawled the Tranco top 30,000 domains. In order to get a web page from a Tranco domain, *protocol://* needs to be prepended to the domain. We used HTTPs as the default protocol; DOMsday used HTTP; TalkGen did not report which protocol was used. Chromium 68, in 2018, started to mark plain HTTP sites as not secure [13]. If advertisement frames containing vulnerabilities are not upgraded to HTTPS, then the mixed content policy in the browser blocks them from loading in HTTPS pages. We observed that our recent Chromium

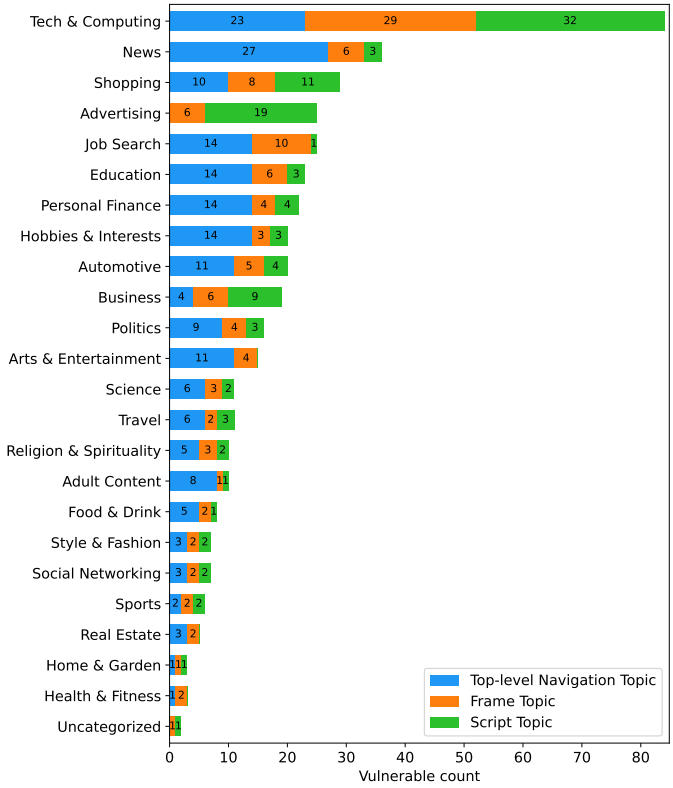


Fig. 11: Vulnerable top URLs, frames and scripts / category.

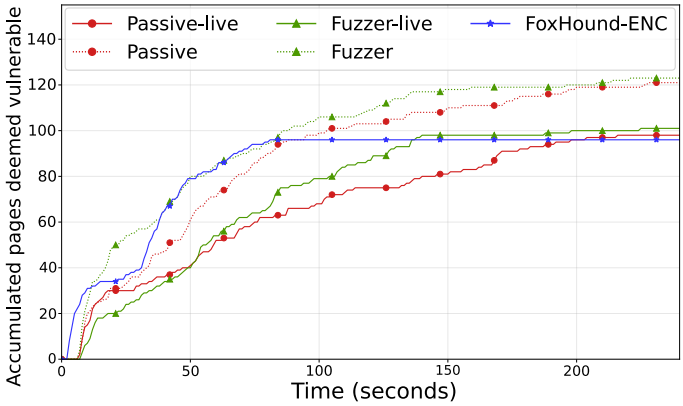


Fig. 12: Accumulated number of pages deemed vulnerable as analysis time increases. Web archiving helps to rediscover more vulnerabilities and reproduce past results.

blocked the loading of 8,967 resources due to Mixed Content policy on our Passive crawl, on 931 different pages (2% of the Core dataset). To further pinpoint the differences between our study and DOMsday, we contacted the authors of DOMsday and obtained a dataset of 849 confirmed flows found by DOMsday in 2018. Passive could confirm only 25 of those. Of the remaining 824, 104 are no longer reachable, and 448 no longer load the vulnerable script. We manually sampled 10 of the remaining 272 cases: In 7 cases, the vulnerable script no longer contained the sink call that it did in 2018, as verified using the Wayback machine [2]. In the other 3 cases, the vulnerable script was considerably different from the 2018 version, and we could not manually find vulnerabilities in the current version.

To conclude, while it is difficult to identify with certainty all the reasons for the disparity between the number of flows across all the studies over the past decade, our investigation strongly suggests that the difference reflects a shift in how browsers handle webpages and web content. URL encoding, now implemented by default on modern browsers, is effective in preventing exploitation of DOM-XSS vulnerabilities. For instance, DOMsday injected their payload on the fragment part of the URL, which at the time was not URL encoded by default in Chromium, but is encoded by recent versions. Along with Ad blocking and increased adoption of HTTPS, the number of DOM-XSS that are exploitable decreased significantly.

V. Discussions

Limitations of the web archive component. To examine the reproducibility of our experiment results using web archiving, we re-analyzed the 194 vulnerable pages 3 months after our original crawl. We ran Passive and Fuzzer using the previously created web archive. We also ran FoxHound-ENC, Passive (Passive-live), and Fuzzer (Fuzzer-live) to analyze live pages without the web archive, using the same timeout as before. Figure 12 summarizes the results. The Y axis is the cumulative number of confirmed vulnerable pages detected, and the X axis is the time taken from opening the browser to eventually find an exploitable flow on each page. Figure 12 shows that using the web archive helps reproduce past vulnerabilities more reliably, as both Passive and Fuzzer rediscovered significantly more vulnerabilities than their counterparts that interacted purely with the live page.

However, our web archiving component fails to reproduce some of the vulnerabilities. There are 6 pages for Passive and 9 for Fuzzer that we no longer find vulnerable even when using the web archive. Some sources of non-determinism (e.g., the use of the Math.random API and different server load causing differences in page loading times) are challenging to handle, even with our web archive's request-matching algorithm and defaulting to request from the live server when no matches are found. To further investigate the effectiveness of our web archive in replaying past resources, we show in Fig. 13 the percentage of responses that were replayed from the web archive compared with the total responses that were served to our browser during fuzzing of pages in the Vulnerable dataset. We only show the results for the 133 pages that Fuzzer previously found vulnerable, and we also include with vertical lines the pages for which our Fuzzer could not rediscover the vulnerabilities. For 7 of the 9 pages that our Fuzzer no longer finds vulnerable, at least one resource was served from the live page. Those new responses from the live server may not be vulnerable anymore, may not exist (returning 404) or may have significantly changed since we first crawled the page. In 2 of

the 9 cases, even though our web archive contained everything that was requested from the browser, that was not enough to rediscover the vulnerability. This suggests that the rediscovery of some vulnerabilities depends not only on the availability of previously archived resources but also on specific run-time conditions or interactions that are still not properly replicated with our web archiving solution.

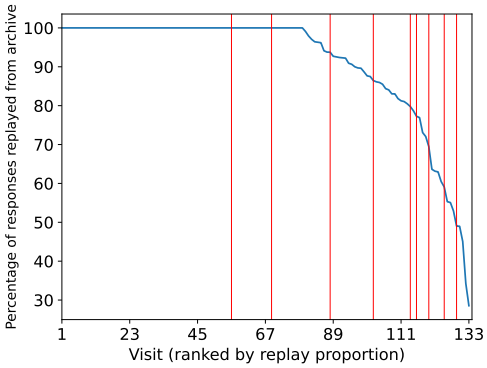


Fig. 13: Percentage of responses that were replayed from the web archive for each vulnerable page found by the Fuzzer. Vertical red lines indicate pages for which Fuzzer could not rediscover the vulnerability 3 months after the original crawl.

Trusted Types. Trusted Types (TT) is a browser security feature that intends to protect clients from DOM-XSS by preventing the injection of untrusted data into vulnerable sinks. TT allows a developer to define a policy, specifying transformations of input that will flow to a sink to make it safe. TT then ensures that either the transformation is applied to the input before being passed to the sink, or the execution is blocked. TT is not a complete solution to DOM-XSS, as it depends on the policy defined by the developers. Though we do not have precise data on how many pages in our dataset implemented TT, data provided by Chrome Platform Status suggest that around 13.3% of the pages enforce TT [1], up from 5.8% since TalkGen launched their crawl.

VI. Related Work

We discuss additional related work in DOM-XSS detection and exploring execution paths in JavaScript programs.

DOM-XSS detection infrastructures. Another way to identify DOM-XSS vulnerabilities is to instrument the JavaScript resources to implement dynamic taint analysis. In comparison, using an instrumented browser for taint tracking has several advantages, including lower time and memory overhead and avoiding the need to bypass page integrity checks [25, 58]. Unfortunately, instrumenting browsers also has a few disadvantages. For example, the instrumentation may not be resilient to browser updates [24]. Consequently, it is extremely difficult for academic researchers to keep up with the Chromium updates. More importantly, direct comparison with DOMsday's taint-enabled browser is unfeasible since DOMsday uses Chromium 54, released in 2016, which does not support recent ECMA versions used by web pages. TalkGen's taint-enabled browser is open-source and actively maintained, but it is based on Firefox, which does not have internal mechanisms to keep track of JavaScript coverage, unlike Chromium. All SWIPE components use Chromium 126 as the base browser, upgraded from DOMsday's original browser with substantial engineering effort.

Detecting other client-side JavaScript vulnerabilities. As client-side code has increased in complexity, vulnerabilities such as client-side prototype pollution [22, 23] and client-side CSRF [27] have increasingly become an issue. While client-side CSRF can be detected using similar techniques as DOM-XSS vulnerabilities, and our instrumented browser can be easily extended to include more sinks, prototype pollution has a more complex pattern and requires more effort to detect and confirm compared to DOM-XSS vulnerabilities. Existing work [22, 23] has leveraged code-property-graph-based approaches for detection. Automatic confirmation of these vulnerabilities is far more complex than confirming DOM-XSS vulnerabilities. As a result, prior work resorted to manual confirmation.

Path exploration in JavaScript. Symbolic execution and fuzzing are widely used for program exploration [10, 32, 53]. Several works implement symbolic execution for JavaScript [33, 52, 62]. Closest to our work is ExpoSE [33], which uses dynamic symbolic execution (DSE) to explore website JavaScript code. ExpoSE focuses on modeling JavaScript regular expression semantics in order to increase code coverage. Our work, like ExpoSE, involves the creation of symbolic models for JavaScript operations. However, we focus on modeling string operations that are commonly used in URL component processing in order to synthesize GET parameters and fragment values, both key vectors for DOM-XSS [36]. We additionally focus on the efficiency of DSE through SMT rewriting and portfolio solving (Sec. III-C). While DSE still faces scalability issues, its performance could be enhanced not only by improving symbolic models of common JavaScript functions but possibly by prioritizing constraint solving based on vulnerability classification of the functions involved in the relevant program paths. Melicher et al. [37] previously showed that such classification is feasible via static analysis using deep neural network-based approaches.

More scalable but less effective approaches, such as fuzzing tools [3, 20, 60], share our DSE component's goal of finding GET parameters but typically rely on enumerating values from a *wordlist* or simple heuristics. ffuf [20] is a web fuzzer that can automatically uncover GET parameters via fuzzing. A user defines a URL template of the form `https://..path?{fuzz}=val` and ffuf attempts to enumerate values of `fuzz` that lead to a valid response. Similarly, wfuzz [3] enumerates candidate values for a specified `fuzz` placeholder placed in an HTTP request component, such as a query parameter. Unlike ffuf and wfuzz, our DSE relies on SMT formulae whose solutions are assignments to GET parameters that satisfy constraints from the page. This reduces

DSE's search space to find page-specific GET parameters that often cannot be found in ffuf's or wfuzz's default wordlists. Finally, Wapiti is a web scanner that can statically discover GET parameters from a target webpage, usually by inspecting links or forms on the page. Parameters found with this method may not always produce different client-side behavior when included in the URL. In contrast, GET parameters and fragment values found by DSE originate from concrete conditionals in the client code.

Simulating user interactions. Existing web page fuzzers for generating user interactions often limit themselves to filling forms and simulating clicks [7, 34, 35, 56, 57]. CrawlJax [38] and JÅk [48] have similar limitations, although they can fire other event handlers via JavaScript. Programmatically firing event handlers may result in false positives, as there might be no way for a real user to execute an event handler, e.g., the associated DOM element might be invisible. To the best of our knowledge, ours is the first fuzzing work to support a wide range of realistic interactions with extensive support for 55 event handlers (complete list in our tech report [50]). Recently, LOAD-AND-ACT [61] studied realistic simulation of user inputs, though limited to mouse and keyboard events. Their approach emphasizes sophisticated search strategies to simulate user actions, but these strategies, although capable of deep frontend analysis, are computationally expensive. In contrast, SWIPE's Fuzzer scales effectively, enabling evaluations on thousands of webpages. Additionally, SWIPE supports a much broader range of user actions. One example is the `onresize` event handler, which Appendix A-A identifies as significant for DOM-XSS detection due to frequent inclusion of sink calls.

VII. Conclusion

This paper introduced SWIPE, a novel DOM-XSS detection infrastructure. SWIPE has a fuzzing component that simulates user interaction and triggers event handlers. This capability improves the number of confirmed flows over prior work by 15%. Second, SWIPE leverages symbolic execution to find pages' GET parameters. Together, these capabilities improve the number of confirmed flows over previous work by 43%. Finally, we discussed how the parallel evolution of web content (e.g., JavaScript versions, page composition) and browsers (e.g., URL-encoding variants) incurs difficult challenges—many of which we overcome—for replicating prior DOM-XSS measurements.

VIII. Ethics Considerations

In this section, we discuss ethical considerations pertinent to our research, focusing specifically on mitigating potential harm during analysis of web pages and ensuring responsible practices throughout our experiments.

Avoiding risks during vulnerability confirmation and other evaluations. Similarly to TalkGen, we inject payloads both in the GET parameters and hash fragment values, but our payloads are instead designed as markers containing special characters and are explicitly constructed not to execute JavaScript code or alter the behavior of the websites. With respect to our evaluation and comparative analysis against existing off-the-shelf tools for GET parameter fuzzing, we deliberately refrained from actively executing these tools, as doing so could result in thousands of automated requests potentially burdening web servers. Instead, our evaluation approach involved examining the wordlists utilized by these tools, thereby completely eliminating the risk of unintended server impact.

Responsible disclosure. Across all novel components evaluated we found DOM-XSS confirmed flows in 194 pages of the Core dataset. We followed Khodayari et al. [26] methodology to discover security contacts of vulnerable domains and responsibly disclosed all vulnerabilities. So far, we got one reply; it acknowledged and committed to patch the vulnerability.

Acknowledgment

This work was partially supported by Carnegie Mellon CyLab and by FCT/MECI through national funds and when applicable co-funded EU funds (UIDB/50008/2020, Instituto de Telecomunicações, and UIDB/50021/2020, INESC-ID multi-annual funding, and PhD grant SFRH/BD/150692/2020).

References

[1] Chrome platform status. https://chromestatus.com/metrics/feature/timeline/popularity/3160. accessed 2025-07-21.

[2] Wayback machine, 1996–. https://web.archive.org.

[3] Wfuzz – the web fuzzer. https://github.com/xmendez/wfuzz, 2014.

[4] WARC, web ARChive file format, 2022–. https://www.loc.gov/preservation/digital/formats/fdd/fdd000236.shtml.

[5] Souphiane Bensalim, David Klein, Thomas Barber, and Martin Johns. Talking about my generation: Targeted DOM-based XSS exploit generation using dynamic data flow analysis. In *EuroSec*, 2021.

[6] Nikolaj Bjørner, Leonardo de Moura, Lev Nachmanson, and Christoph M. Wintersteiger. Programming Z3. Lecture Notes in Computer Science. 2019.

[7] Kostas Drakonakis, Sotiris Ioannidis, and Jason Polakis. Rescan: A middleware framework for realistic and robust black-box web application scanning. In *Proc. of NDSS*, 2023.

[8] ECMA International. ECMA-262 – ECMAScript language specification. https://tc39.es/ecma262/, 2024.

[9] Ayush Goel, Jingyuan Zhu, Ravi Netravali, and Harsha V. Madhyastha. Jawa: Web archival in the era of JavaScript. In *Proc. of ODSI*, 2022.

[10] Brandon Wen Heng Goh. American fuzzy lop (AFL). 2019.

[11] Google. Expanding user protections on the web. https://blog.chromium.org/2017/11/expanding-user-protections-on-web.html, 2017.

[12] Google. Further protections from harmful ad experiences on the web. https://blog.chromium.org/2018/11/further-protections-from-harmful-ad.html, 2018.

[13] Google. A secure web is here to stay. https://security.googleblog.com/2018/02/a-secure-web-is-here-to-stay.html, 2018.

[14] Google. Under the hood: How Chrome's ad filtering works. https://blog.chromium.org/2018/02/how-chromes-ad-filtering-works.html, 2018.

[15] Google. Building a more private web: A path towards making third party cookies obsolete. https://blog.chromium.org/2020/01/building-more-private-web-path-towards.html, 2020.

[16] Google. Protecting against resource-heavy ads in Chrome. https://blog.chromium.org/2020/05/resource-heavy-ads-in-chrome.html, 2020.

[17] Google. Samesite cookie changes in february 2020: What you need to know. https://blog.chromium.org/2020/02/samesite-cookie-changes-in-february.html, 2020.

[18] J. Gruber and the V8 Project. Block code coverage. Google Docs, 2018. Design document for V8 code-coverage support including block-level instrumentation.

[19] Florian Hantke, Stefano Calzavara, Moritz Wilhelm, Alvise Rabitti, and Ben Stock. You call this archaeology? evaluating web archives for reproducible web security measurements. In *Proc. of CCS*, 2023.

[20] Joona Hoikkala. ffuf – fuzz faster u fool. https://github.com/ffuf/ffuf, 2018.

[21] IAB Technology Laboratory, Inc. Content taxonomy 3.0 and descriptive vectors. https://github.com/InteractiveAdvertisingBureau/Taxonomies/blob/develop/Taxonomy%20Mappings/Content%201.0%20to%20Ad%20Product%202.0.tsv, June 2022.

[22] Zifeng Kang, Song Li, and Yinzhi Cao. Probe the proto: Measuring client-side prototype pollution vulnerabilities of one million real-world websites. In *Proc. of NDSS*, 2022.

[23] Zifeng Kang, Muxi Lyu, Zhengyu Liu, Jianjia Yu, Runqi Fan, Song Li, and Yinzhi Cao. Follow my flow: Unveiling client-side prototype pollution gadgets from one million real-world websites. In *Proc. of IEEE SP*, 2025.

[24] Rahul Kanyal and Smruti R Sarangi. PanoptiChrome: A modern in-browser taint analysis framework. In *Proc. of WebConf*, 2024.

[25] Rezwana Karim, Frank Tip, Alena Sochuurkov'a, and Koushik Sen. Platform-independent dynamic taint analysis for JavaScript. *Proc. of TSE*, 2018.

[26] Soheil Khodayari, Thomas Barber, and Giancarlo Pellegrino. The great request robbery: An empirical study of client-side request hijacking vulnerabilities on the web. In *Proc. of IEEE SP*, 2024.

[27] Soheil Khodayari and Giancarlo Pellegrino. JAW: Studying client-side CSRF with hybrid property graphs and declarative traversals. In *Proc of USENIX*, 2021.

[28] Amit Klein. DOM based cross site scripting or XSS of the third kind. http://www.webappsec.org/projects/articles/071105.shtml, 2005.

[29] David Klein, Thomas Barber, Souphiane Bensalim, Ben Stock, and Martin Johns. Hand sanitizers in the wild: A large-scale study of custom JavaScript sanitizer functions. In *Proc. of IEEE SP*, 2022.

[30] Victor Le Pochat, Tom Van Goethem, Samaneh Tajalizadehkhoob, Maciej Korczyński, and Wouter Joosen. Tranco: A research-oriented top sites ranking hardened against manipulation. In *Proc. of NDSS*, 2019.

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

[32] Jun Li, Bodong Zhao, and Chao Zhang. Fuzzing: a survey. *Cybersecurity*, 1:1–13, 2018.

[33] Blake Loring, Duncan Mitchell, and Johannes Kinder. ExpoSE: Practical symbolic execution of standalone JavaScript. In *Proc. of SPIN*, 2017.

[34] marmelab. Gremlins.js. https://github.com/marmelab/gremlins.js, 2020.

[35] S McAllister, E Kirda, and C Kruegel. Expanding human interactions for in-depth testing of web applications. In *Proc. of RAID*, 2008.

[36] William Melicher, Anupam Das, Mahmood Sharif, Lujo Bauer, and Limin Jia. Riding out domsday: Towards detecting and preventing DOM cross-site scripting. In *Proc. of NDSS*, 2018.

[37] William Melicher, Clement Fung, Lujo Bauer, and Limin Jia. Towards a lightweight, hybrid approach for detecting DOM XSS vulnerabilities with machine learning. In *Proc. of WebConf*, 2021.

[38] Ali Mesbah, Arie Van Deursen, and Stefan Lenselink. Crawling Ajax-based web applications through dynamic analysis of user interface state changes. *ACM Transactions on the Web*, 2012.

[39] Mozilla. Firefox 86 introduces total cookie protection. https://blog.mozilla.org/security/2021/02/23/total-cookie-protection/, 2021.

[40] Mozilla. Firefox 87 introduces smartblock for private browsing. https://blog.mozilla.org/security/2021/03/23/introducing-smartblock/, 2021.

[41] Mozilla. Firefox 90 introduces smartblock 2.0 for private browsing. https://blog.mozilla.org/security/2021/07/13/smartblock-v2/, 2021.

[42] Mozilla. Firefox 93 features an improved smartblock and new referrer tracking protections. https://blog.mozilla.org/security/2021/10/05/firefox-93-features-an-improved-smartblock-and-.new-referrer-tracking-protections/, 2021.

[43] Mozilla. Firefox rolls out total cookie protection by default to more users worldwide. https://blog.mozilla.org/en/mozilla/firefox-rolls-out-total-cookie-protection-.by-default-to-all-users-worldwide/, 2022.

[44] National Institute of Standards and Technology (NIST). NVD vulnerability statistics for XSS - last 3 months. https://nvd.nist.gov/vuln/search/statistics, 2025.

[45] OWASP Foundation. OWASP zed attack proxy (ZAP). https://www.zaproxy.org/. Accessed: 2025-07-21.

[46] OWASP Foundation. Cross site scripting (XSS). https://owasp.org/www-community/attacks/xss/, 2024.

[47] Inian Parameshwaran, Enrico Budianto, Shweta Shinde, Hung Dang, Atul Sadhu, and Prateek Saxena. DexterJS: Robust testing platform for DOM-based XSS vulnerabili-

ties. In *Proc. of FSE*, 2015.
[48] Giancarlo Pellegrino, Constantin Tschürtz, Eric Bodden, and Christian Rossow. jÄk: Using dynamic analysis to crawl and test modern web applications. In *Proc. of RAID*, 2015.
[49] Andrew Reynolds, Andres Nötzli, Clark Barrett, and Cesare Tinelli. High-level abstractions for simplifying extended string constraints in SMT. In *Proc. of CAV*, Lecture Notes in Computer Science, 2019.
[50] Nuno Sabino, Darion Cassel, Rui Abreu, Pedro Adão, Lujo Bauer, and Limin Jia. DOM-XSS detection via webpage interaction fuzzing and URL component synthesis (technical report). *Carnegie Mellon Kilthub*, 2025. DOI:10.1184/R1/30010783.
[51] Nuno Sabino, Darion Cassel, Rui Abreu, Pedro Adão, Lujo Bauer, and Limin Jia. SWIPE: DOM-XSS analysis infrastructure. https://doi.org/10.5281/zenodo.15883603, July 2025.
[52] José Fragoso Santos, Petar Maksimović, Théotime Grohens, Julian Dolby, and Philippa Gardner. Symbolic Execution for JavaScript. In *Proc. of PPDP*, 2018.
[53] Edward J Schwartz, Thanassis Avgerinos, and David Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *Proc. of IEEE SP*, 2010.
[54] Koushik Sen, Swaroop Kalasapur, Tasneem Brutch, and Simon Gibbs. Jalangi: A selective record-replay and dynamic analysis framework for JavaScript. In *Proc. of FSE*, 2013.
[55] Koushik Sen and Manu Sridharan. Jalangi2, 2014–. https://github.com/Samsung/jalangi2.
[56] Orpheas van Rooij, Marcos Antonios Charalambous, Demetris Kaizer, Michalis Papaevripides, and Elias Athanasopoulos. webfuzz: Grey-box fuzzing for web applications. In *Proc. of ESORICS*, 2021.
[57] Wenhua Wang, Sreedevi Sampath, Yu Lei, and Raghu Kacker. An interaction-based test sequence generation approach for testing web applications. In *Proc. of IEEE HASE*, 2008.
[58] Zilun Wang, Wei Meng, and Michael R Lyu. Fine-grained data-centric content protection policy for web applications. In *Proc. of CCS*, 2023.
[59] Wapiti Scanner Project. Wapiti: Web-application vulnerability scanner. https://wapiti-scanner.github.io/, 2024. Version 3.2.0, accessed 2025-07-21.
[60] Sunny Wear. *Burp Suite Cookbook: Practical recipes to help you master web penetration testing with Burp Suite*. Packt Publishing Ltd, 2018.
[61] Nico Weidmann, Thomas Barber, and Christian Wressnegger. Load-and-act: Increasing page coverage of web applications. In *Proc. of ISC*, 2023.
[62] Feng Xiao, Jianwei Huang, Yichang Xiong, Guangliang Yang, Hong Hu, Guofei Gu, and Wenke Lee. Abusing hidden properties to attack the Node.js ecosystem. In *Proc of USENIX*, 2021.
[63] Xinshi Zhou and Bin Wu. Web application vulnerability fuzzing based on improved genetic algorithm. In *Proc. of ITNEC*, 2020.

APPENDIX A
APPENDIX

A. Executed event handlers

We show in Fig. 14 how the number of executed event handlers compares between Passive and Fuzzer for the Core dataset. We only show the top 10 supported event handlers when sorted by the frequency of presence of sink calls in the code of the handler. The `mouseenter` event handler is the event handler type that has the most sink calls among all event handlers. Our fuzzer may not be able to trigger an event handler due to a variety of reasons, including: needing complex input (e.g., on a textbox) where symbolic execution could be more appropriate; needing a complex sequence of interactions to execute; frame with the event handler may have been removed from the DOM during navigation.

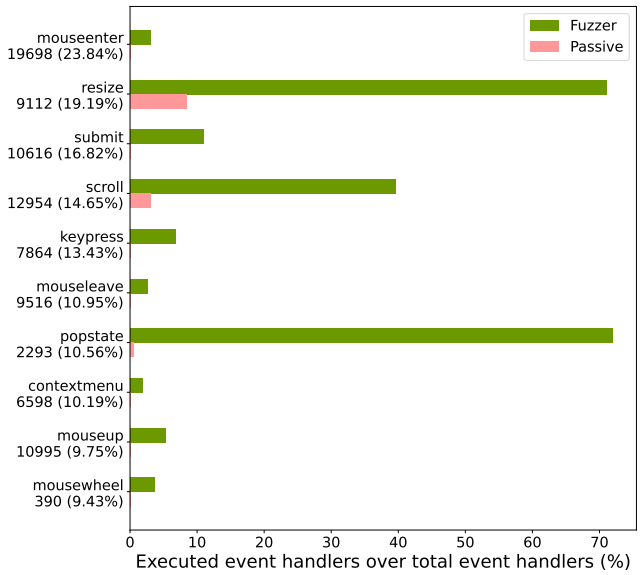


Fig. 14: Percentage of event handlers executed by Passive and Fuzzer, over the total event handlers present on the pages from the Core dataset, for the 10 supported event handlers that have higher sink calls frequency. Beneath each event handler name we also show how many have calls to sinks and the percentage (number of event handlers with sink calls divided by total number of event handlers of that type)

Additionally, we show in Fig. 15 the same data but only within the Vulnerable dataset, where we also run our simpleFuzzer that does not combine actions. That figure shows a clear improvement in executed event handlers when we let the Fuzzer combine actions, highlighting the importance of creating groups of HLAs and exchanging HLAs between those groups (cross-over). Another interesting property illustrated in Fig. 15 is that `focusout` and `focusin` are the event handlers that have the most number of calls to sinks in pages from the Vulnerable dataset. Those events were not even in the top 10

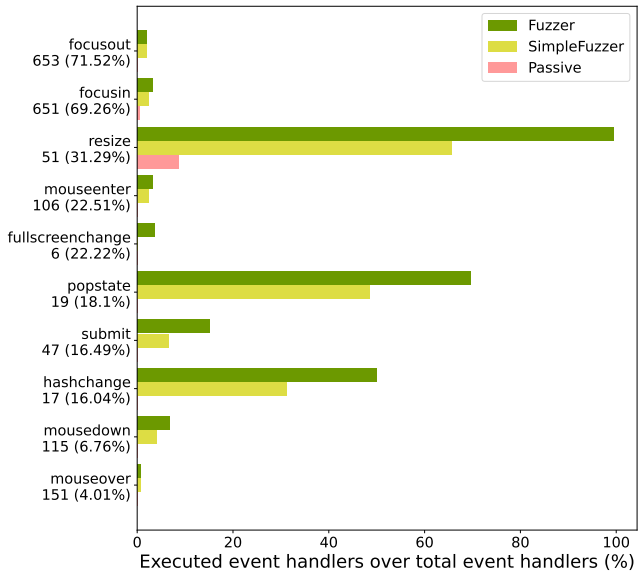


Fig. 15: Percentage of event handlers in the Vulnerable dataset that were executed by Passive, Fuzzer and the simpleFuzzer that does not combine actions, for the 10 supported event handlers with higher sink calls frequency. This figure is analogous to Fig. 14, but limited to the Vulnerable dataset.

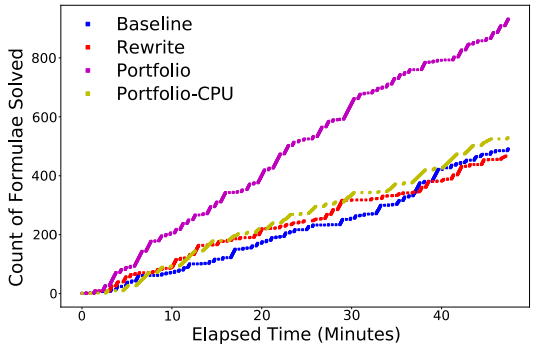


Fig. 16: Results of the portfolio solving evaluation.

of event handlers with most sink calls for the Core dataset, showing an interesting bias for vulnerable pages (or at least, for pages that SWIPE +FoxHound find vulnerable).

Furthermore, we report that event handlers constitute 1.56% of all JavaScript code found in pages in our crawl. With respect to **supported** event handlers, they constitute 1.11% of all JavaScript code we saw. This means that our fuzzer can theoretically simulate user interactions for 71% of all event handler code that we found in the wild. Also note that this is not inconsistent with the 13% additional JavaScript code executed with the Fuzzer compared with Passive, as an event handler may contain calls to other functions or can even load extra JavaScript resources.

B. Efficacy of SMT rewriting in portfolio

We evaluate the performance of the portfolio solver on a set of SMT queries generated by DSE on real webpages in the Tranco top 1000 [30]. From a set of 104,000 SMT formulae selected from prior runs of our DSE analysis, we selected 1,000 SMT formulae uniformly at random. The formulae were not necessarily satisfiable, and some cannot be solved within the time of any evaluated solvers. We ran the experiment with three conditions: the *Baseline* solver, which utilizes Z3 in its default configuration with no preprocessing of the SMT formulae; the *Rewrite* solver, which utilizes the SMT rewriting algorithm; and the *Portfolio* solver, which utilizes both Baseline and Rewrite solving in parallel. All solvers were run with a timeout of 25 seconds, and were evaluated on the basis of the number of SMT formulae that Z3 returned either `sat` or `unsat`, within the timeout. To account for the fact that the portfolio solver runs two solvers in parallel, thus using twice the CPU resources of the baseline and rewrite solvers, we also include a fourth condition, *Portfolio-CPU*, which is the same as the portfolio solver, but counts the elapsed time as the sum of the elapsed time of the two parallel solvers.

The results of the experiment are shown in Fig. 16. The y-axis is a count of the number of SMT formulae that were solved. The x-axis is the time in minutes; it extends until 47 minutes, which was the time taken by the fastest solver to run on its set of SMT formulae. The Baseline solver solved 491 SMT formulae within the experiment duration. The Rewrite solver solved 467, representing a 5% regression compared to the baseline solver. However, the Baseline and Rewrite solvers are very close; for much of the experiment duration, the Rewrite solver was performing better than the Baseline. The portfolio solver performed significantly better (90% improvement over Baseline), solving 932 SMT formulae within the experiment duration. Even after accounting for total CPU time used, we find that the portfolio solver still performs 8% better than the baseline solver, solving 529 SMT formulae.

Input	Chromium 54	Chromium 126
window.location.href	Partial	Yes
window.location.hash	No	Yes
document.referrer	Partial	Yes
window.location.search	Yes	Yes

TABLE VI: URL encoding differences between our browser version (Chromium 126) and the one used by DOMsday. We found no differences between ours and the latest version.

C. URL encoding by browsers

We show in Tab. VI the differences we found in URL encoding between our browser version (Chromium 126) and the one used by DOMsday (Chromium 54). The main difference shown in the table appears in the encoding of the `window.location.hash` property. In Chromium 126, the fragment in the URL is URL encoded, whereas in Chromium 54 it is not. This has an impact on the flows that can be confirmed in each specific browser version, since one of the steps in the confirmation methodology is checking whether the tainted parts of the final sink argument are not URL encoded.

APPENDIX B
ARTIFACT APPENDIX

In this section we describe how to download, install and run SWIPE, so that other researchers can use our components for DOM-XSS detection.

A. Description & Requirements

1) How to access: SWIPE is available in the following link: https://doi.org/10.5281/zenodo.15883603.

2) Hardware dependencies: 4GB RAM, 4 cores and at least 30GB of storage.

3) Software dependencies: Tested operating system: Linux x86. Required software: Docker ($\geq$ version 27), Tiger VNC or other VNC viewer.

4) Benchmarks: The experiments in this paper involved the collection of the top 30,000 Tranco domains[2] generated on 04 March 2025. We crawled each of those 30,000 domains and extracted a maximum of 5 subpages from each. We call this our *core* dataset.

B. Artifact Installation & Configuration

In this section we describe the necessary steps to install SWIPE. All these steps are also included in the README.md file present in the provided link.

Installation with Docker

Open a terminal in the root of this project (where the Dockerfile is located). Build the SWIPE image using the following command:

```
$ docker build -t swipe:latest .
```

This takes around 10 minutes on a machine with 8 cores and 32GB RAM. Once it is done, you can confirm the image was built with the following command:

```
$ docker image ls
REPOSITORY ...
swipe ...
```

C. Experiment Workflow

The high-level workflow when using SWIPE is as follows:

1) A target webpage is selected. A public URL must be known.
2) SWIPE is configured in a Docker container to run Passive (default), Fuzzer or DSE components, while archiving (default) or replaying a previously created archive.
3) SWIPE is invoked on the target webpage. The enabled component navigates to the page and analyses it. Flow results are parsed.
4) Optionally, SWIPE is invoked on a set of URLs that are crafted in such a way that allows for confirming vulnerabilities.

In our crawl, to run an experiment over a set of URLs, we used one container instance for each page, and then aggregated the results across pages. In this artifact evaluation we will use the same container for all experiments.

[2]Available at https://tranco-list.eu/list/3NPQL.

D. Major Claims

The results provided in this paper can only be reproduced by performing a large scale evaluation over 30,000 Tranco domains. This artifact instead provides a way to evaluate the functionality of SWIPE components, based on the following underlying claims made in the paper:

- (C1): SWIPE uses symbolic execution to automatically discover and synthesize URL parameters that lead to new execution paths and potential vulnerabilities. This is supported by experiments (E1) and (E2).
- (C2): SWIPE's fuzzer triggers event handlers and uncovers vulnerabilities that only manifest through user interaction. This claim is supported by experiments (E1) and (E3).
- (C3): SWIPE's web archive increases the stability of the target webpage. Evidence for this can be obtained with experiment (E4).

E. Evaluation

Initial setup. The following experiments will be performed on a single container using SWIPE's docker image. To run the container, issue the following command:

```
$ docker run --rm -it -p 5550:5550 \
   --entrypoint=bash swipe
```

In order to evaluate the functionality of the web archive (E4), a XVFB server must be started on the container to listen to VNC connections. To do that, run the following command in the container:

```
./jalangi2-workspace/run_xvfb.sh
```

You can now use a VNC viewer to connect to http://localhost:5550 with the password *DEBUG*.

To finish the setup, go to SWIPE's main folder in the container. Every command in the following experiments is supposed to be executed on that folder.

```
cd jalangi2-workspace/scripts/swipe/
```

Example webpage. In experiments (E1)–(E3) SWIPE will be launched against an example webpage[3]. This webpage has 3 DOM-XSS vulnerabilities: one that is triggered during page initialization (E1); another that requires the GET parameter value to include a specific string (E2); and a third DOM-XSS that is triggered when the mouse is scrolled while over a certain element of the page (E3).

[3]The page is available here http://swipeexample.s3-website-eu-west-1.amazonaws.com/example_page.html?gp

1) Experiment (E1): [Passive] [10 human-minutes + 3 compute-minutes]: In this experiment, SWIPE will replicate passive navigation when analyzing a page. This will serve as a baseline for subsequent experiments.

[Preparation] None. SWIPE runs Passive by default and it is already configured to run against the example page.

[Execution] Run SWIPE by issuing the command below:

```
$ ./run.sh
```

[Results] The file `./output.txt` should indicate that one potential flow was discovered by SWIPE, by containing the following:

```
... 'function': 'vulnerable_passive', 'col'
    ↪ : 22, 'lineno': 22 ...
```

This corresponds to the DOM-XSS vulnerability that is triggered during page initialization.

2) Experiment (E2): [DSE] [2 human-minutes + 6 compute-minutes]: In this experiment, SWIPE will use symbolic execution to synthesize GET parameters that will explore more JavaScript program paths in the example page.

[Preparation] This can be done right after running (E1). DSE must be enabled by setting the following flag in SWIPE's configuration file `config/config.json`:

```
    "try-alternative-paths": true
```

[Execution] Run SWIPE again.

```
$ ./run.sh
```

[Results] The file `./output.txt` should indicate that an extra potential flow was discovered by SWIPE, by having the following content:

```
... 'function': 'vulnerable_dse' ...
```

This corresponds to a vulnerable function that is only called when the GET parameters satisfy specific constraints.

3) Experiment (E3): [Fuzzer] [2 human-minutes + 3 compute-minutes]: In this experiment, SWIPE will use fuzzing to interact with the example page and discover a vulnerability that was not previously discovered.

[Preparation] This can be done after (E1). Make sure DSE is disabled and the Fuzzer is enabled, by ensuring the following configuration in the `config/config.json` file:

```
    "try-alternative-paths": false
    "run-ui-fuzzer": true
```

[Execution] Run SWIPE once again.

```
$ ./run.sh
```

[Results] The file `./output.txt` should indicate that an extra potential flow was discovered by SWIPE, by having the following content:

```
... 'function': 'vulnerable_fuzzer', 'col':
    ↪  26, 'lineno': 7 ...
```

This corresponds to a vulnerable function that is only called when a certain user interaction is performed on the page.

4) Experiment (E4): [Web Archive] [4 human-minutes + 10 compute-minutes]: In this experiment, SWIPE will create an archive for a webpage that displays random images and then replay the archive, serving the same image consistently to the browser.

[Preparation] For this experiment, disable both the Fuzzer and DSE in SWIPE's configuration file `config/config.json`. Also, make sure that SWIPE is in archiving mode:

```
    "try-alternative-paths": false
    "run-ui-fuzzer": false
    "mitmproxy-archivemode": true
```

Finally, set the target page by replacing the contents of the `config/sample_targets` with https://randomwordgenerator.com/picture.php. That will point SWIPE to that page.

[Execution] Run SWIPE yet again and pay attention to the image that is going to be loaded by the browser, in the VNC viewer window.

```
$ ./run.sh
```

Once the browser closes, configure the web archive to be in replay mode, by making the following changes in `config/config.json`:

```
    "mitmproxy-archivemode": false
    "mitmproxy-replaymode": true
```

Once you run SWIPE again, SWIPE will replay from the previously created web archive and thus, the previously seen image will be loaded once again, as can be observed in the VNC viewer.

```
./run.sh
```

More information can be found in the artifact [51].