JAW: Studying Client-side CSRF with Hybrid Property Graphs and Declarative Traversals

Soheil Khodayari
CISPA Helmholtz Center for Information Security

Giancarlo Pellegrino
CISPA Helmholtz Center for Information Security

Abstract

Client-side CSRF is a new type of CSRF vulnerability where the adversary can trick the client-side JavaScript program to send a forged HTTP request to a vulnerable target site by modifying the program's input parameters. We have little-to-no knowledge of this new vulnerability, and exploratory security evaluations of JavaScript-based web applications are impeded by the scarcity of reliable and scalable testing techniques. This paper presents JAW, a framework that enables the analysis of modern web applications against client-side CSRF leveraging declarative traversals on *hybrid property graphs*, a canonical, hybrid model for JavaScript programs. We use JAW to evaluate the prevalence of client-side CSRF vulnerabilities among all (i.e., 106) web applications from the Bitnami catalog, covering over 228M lines of JavaScript code. Our approach uncovers 12,701 forgeable client-side requests affecting 87 web applications in total. For 203 forgeable requests, we successfully created client-side CSRF exploits against seven web applications that can execute arbitrary server-side state-changing operations or enable cross-site scripting and SQL injection, that are not reachable via the classical attack vectors. Finally, we analyzed the forgeable requests and identified 25 request templates, highlighting the fields that can be manipulated and the type of manipulation.

1 Introduction

Client-side Cross-Site Request Forgery (client-side CSRF) is a new breed of CSRF vulnerabilities affecting modern web applications. Like the more traditional CSRF, with a brief visit to a malicious URL, an adversary can trick the victim's browser into sending an authenticated security-sensitive HTTP request on the user's behalf towards a target web site without user's consent or awareness. In the traditional CSRF, the vulnerable component is the server-side program, which cannot distinguish whether the incoming authenticated request was performed intentionally, also known as the *confused deputy problem* [45, 55]. CSRF is typically solved by adding a pseudo-random unpredictable request parameter, preventing forgery (see, e.g., [34]), or by changing the default browsers' behavior and avoiding the inclusion of HTTP cookies in cross-site requests (see, e.g., [28, 29]). In the client-side CSRF, the vulnerable component is the JavaScript program instead, which allows an attacker to generate arbitrary requests by modifying the input parameters of the JavaScript program. As opposed to the traditional CSRF, existing anti-CSRF countermeasures (see, e.g., [28, 29, 34]) are not sufficient to protect web applications from client-side CSRF attacks.

Client-side CSRF is very new—with the first instance affecting Facebook in 2018 [24]—and we have little-to-no knowledge of the vulnerable behaviors, the severity of this new flaw, and the exploitation landscape. Studying new vulnerabilities is not an easy task, as it requires the collection and analysis of hundreds of web pages per real web applications. Unfortunately, such analyses are primarily impeded by the scarcity of reliable and scalable tools suitable for the detection and analysis of vulnerable JavaScript behaviors.

In general, studying client-side CSRF vulnerabilities in JavaScript-based web applications is not an easy task. First, there is no canonical representation for JavaScript code. Second, JavaScript programs are event-driven, and we need models that capture and incorporate this aspect into the canonical representation. Third, pure static analysis is typically not sufficiently accurate due to the dynamic nature of JavaScript programs [43, 46, 72], and their execution environment [47], calling for hybrid static-dynamic analysis techniques. Finally, JavaScript libraries constitute a noteworthy fraction of code across web pages, and analyzing them repeatedly leads to inefficient models poorly suitable for detecting vulnerabilities.

In this paper, we address these challenges by proposing *hybrid property graphs* (HPGs), a coherent, graph-based representation for client-side JavaScript programs, capturing both static and dynamic program behaviors. Inspired by prior work [91], we use property graphs for the model representation and declarative graph traversals to identify security-sensitive HTTP requests that consume data values from attacker-controllable sources. Also, we present JAW, a framework for the detection of client-side CSRF that, starting from a seed URL, instantiates HPGs by automatically

collecting web resources and monitoring program execution.

Finally, we instantiated JAW against *all* (i.e., 106) web applications of the Bitnami catalog [2] to detect and study client-side CSRF, covering, in total, over 228M lines of JavaScript code over 4,836 web pages. Overall, our approach uncovers 12,701 forgeable client-side requests affecting 87 web applications. For 203 forgeable requests, we successfully created client-side CSRF exploits against seven web applications that can execute arbitrary server-side state-changing operations or enable cross-site scripting and SQL injection, that are not reachable via the classical attack vectors. Finally, we analyzed forgeable requests and identified 25 distinct request templates, highlighting the fields that can be manipulated and the type of manipulation.

To summarize, we make the following main contributions:

- We perform the first systematic study of client-side CSRF, a new variant of CSRF affecting the client-side JavaScript program, and present a taxonomy of forgeable requests considering two features, i.e., request fields, and the type of manipulation.
- We present *hybrid property graphs*, a single and coherent representation for the client-side of web applications, capturing both static and dynamic program behaviors.
- We present JAW, a framework that detects client-side CSRF by instantiating a HPG for each web page, starting from a single seed URL.
- We evaluate JAW with over 228M lines of JavaScript code in 106 popular applications from the Bitnami catalog, identifying 12,701 forgeable requests affecting 87 applications, out of which we created working exploits for 203 requests of seven applications.
- We release the source code of JAW[1] to support the future research effort to study vulnerable behaviors of JavaScript programs.

2 Background

Before presenting JAW, we introduce the client-side CSRF vulnerability and a running example (§2.1). Then, we present the challenges to analyze client-side CSRF vulnerabilities (§2.2). Finally, we give an overview of our approach (§2.3).

2.1 Client-side CSRF

Client-side CSRF is a new category of CSRF vulnerability where the adversary can trick the client-side JavaScript program to send a forged HTTP request to a vulnerable target site by manipulating the program's input parameters. In a client-side CSRF attack, the attacker lures a victim into clicking a malicious URL that belongs to an attacker-controlled web page or an honest but vulnerable web site, which in turn causes a security-relevant state change of the target site.

Impact. Similarly to the classical CSRF, client-side CSRF can be exploited to perform security-sensitive actions on the server-side and compromise the database integrity. Successful CSRF attacks can lead to remote code execution [51, 69], illicit money transfers [69, 93], or impersonation and identity riding [23, 24, 25, 26, 27, 37], to name only a few instances.

Root Causes. Client-side CSRF vulnerabilities originate when the JavaScript program uses attacker-controlled inputs, such as the URL, for the generation of outgoing HTTP requests. The capabilities required to manipulate different JavaScript input sources (e.g., see [60]) are discussed next.

Threat Model. The overall goal of an attacker is forging client-side HTTP requests by manipulating various JavaScript input sources. In this paper, we consider the URL, window name, document referrer, postMessages, web storage, HTML attributes, and cookies, each requiring different attacker capabilities. Manipulating the URL, window name, referrer and postMessages require an attacker able to forge a URL or control a malicious web page. For example, a *web attacker* can craft a malicious URL, belonging to the origin of the honest but vulnerable web site, that when visited by a victim leads to automatic submission of an HTTP request by the JavaScript program of the target site. Alternatively, a web attacker can control a malicious page and use browser APIs to trick the vulnerable JavaScript of the target page to send HTTP requests. For example, a web attacker can use `window.open()` [21] to open the target URL in a new window, send postMessages [81] to the opened window, or set the window name through `window.name` API [20]. Furthermore, a web attacker can manipulate `document.referrer` leveraging the URL of the attacker-controlled web page.

For web storage and HTML attributes, the attacker needs to add ad-hoc data items in the web storage or DOM tree. A web attacker could achieve that assuming the web application offers such functionalities (e.g., by HTTP requests). Similarly, a web attacker with a knowledge of an XSS exploit can manipulate the web storage or DOM tree. Finally, modifying cookies may require a powerful attacker such as a *network attacker*. This attacker can implant a persistent client-side CSRF payload in the victim's browser by modifying cookies (e.g., see [78, 84, 94]), which can lie dormant, and exploited later on to attack a victim. We observe that all attacks performed by the web attacker can be performed by a network attacker too.

Vulnerability. Listing 1 exemplifies a vulnerable script–based on a real vulnerability that we discovered in SuiteCRM–that fetches a shopping invoice with an HTTP request during the page load. First, the program fetches an HTML input field with id `input` (line 1), and then defines an event handler `h` that is responsible for retrieving the price of the invoice with an asynchronous request and populating the `input` with the price (lines 2-9). For asynchronous requests, the function `h` uses YUI library [22], that provides a wrapper `asyncRequest` for the low-level `XMLHttpRequest` browser API. Then, the function `h` is registered as a handler for a cus-

[1] https://github.com/SoheilKhodayari/JAW

Listing 1: Example client-side CSRF vulnerability derived from SuiteCRM.

```
1  var i = document.querySelector('input');
2  async function h(e){
3   var uri = window.location.hash.substr(1);
4   if (uri.length > 0) {
5     let req = new asyncRequest("POST", uri);
6     // Add Synchronizer Token
7     req.initHeader('X-CSRF-TOKEN', token);
8     var price = await req.send();
9     i.value = price;}}
10  i.addEventListener('loadInvoice', h);
...
14  function showInvoicePrice(input_id) {
15   document.getElementById(input_id).dispatchEvent(new
       CustomEvent('loadInvoice', {}));}
16  showInvoicePrice('input');
```

tom event called `loadInvoice`. This event is dispatched by the function `showInvoicePrice` (lines 14-16). The vulnerability occurs (in lines 3-5) when the JavaScript program uses URL fragments to store the server-side endpoint for the HTTP request, an input that can be modified by the attacker.

Attack. Figure 1 shows an example of attack exploiting the client-side CSRF vulnerabilities of Listing 1. First, the attacker prepares a URL of the vulnerable site, by inserting the URL of the target site as URL fragment (step 1). Then, the victim is lured into visiting the vulnerable URL (step 2), as it belongs to an application that the user trusts. Upon completion of the page load (step 3), the JavaScript code will extract a URL from the URL fragment, and send an asynchronous HTTP request towards the target site, which in turn causes a security-relevant state change on the target server.

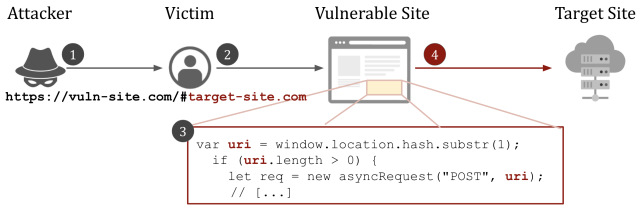
Existing Defenses are Ineffective. Over the past years, the community proposed several defenses against CSRF (e.g., [34, 39, 52, 53, 63, 74]). Recently, browser vendors proposed to introduce a stricter same-site cookies policy [28, 29, 30], by marking all cookies as `SameSite=Lax` by default [90]. Unfortunately, existing mechanisms cannot offer a complete protection against client-side CSRF attacks, e.g., when synchronizer tokens [34, 39] or custom HTTP headers [34, 86] are used, the JavaScript program will include them in the outgoing requests as shown in line 7 of Listing 1. Also, if the browser or the web site is using the same-site policy for cookies, JavaScript web pages, once loaded, can perform preliminar same-site requests to determine whether a pre-established user session exists, circumventing the same-site policy.

2.2 Challenges

In this work, we intend to study the new client-side CSRF vulnerability in the client-side JavaScript code of a web application. Before presenting our solution, we show the challenges we need to address to achieve our objective.

(C1) Static Representational Models. JavaScript programs are incredibly challenging to be analyzed via static analysis. For example, prior work have proposed inter-procedural control flow graphs [50, 67], data flow dependency graphs [62, 82], type analyzers [38, 44, 49], and points-to analysis [61, 83]. Unfortunately, these approaches provide ad-hoc representation of programs, each focusing on an individual aspect that is alone not sufficient to study client-side CSRF. Recently, we have seen new ideas unifying static representations with *code property graphs* (CPGs) [33, 91]. However, these new ideas are not tailored to JavaScript's nuances, such as the asynchronous events [82], or the execution environment [47]. To date, there are no models for JavaScript that can provide a canonical representation to conduct both detection and exploratory analysis of the code.

Figure 1: Example of client-side CSRF attack.



(C2) Vulnerability-specific Analysis Tools. Over the past years, there have been a plethora of approaches to detect vulnerabilities in client-side JavaScript programs. To date, these approaches have been mainly applied to XSS [60, 64, 75, 81, 84], or logic and validation vulnerabilities [35, 36, 66, 76, 79, 80, 87, 89], resulting in tools that are rather tightly *coupled* with the specific analysis of the vulnerability. Thus, researchers seeking to study new client-side vulnerabilities like client-side CSRF are forced to reimplement those approaches rediscovering tweaks and pitfalls.

(C3) Event-based Transfer of Control. Existing unified representations such as CPGs [33, 91] assume that the transfer of control happens *only* via function calls, an assumption no longer valid for JavaScript. In JavaScript, the transfer of control happens also via events which either originate from the environment, e.g., mouse events, or are user-defined, as shown in Listing 1. When an event is dispatched, one or more registered functions are executed, which can change the state of the program, register new handlers, and fire new events. Representing the transfer of control via event handlers is fundamental for the analysis of JavaScript programs.

(C4) Dynamic Web Execution Environment. JavaScript programs rely on many dynamic behaviors that make it challenging to study them via pure static analysis. A typical example is the dynamic code loading [46]. In essence, JavaScript programs can be streamed to the user's web browser, just like other resources. Thus, contrary to the assumption in most static analysis approaches, the entire JavaScript code may not be available for the analysis [43]. Another example is the interaction between JavaScript and the DOM tree. Consider, for example, two variables containing the same DOM tree node; however, the content of one variable is fetched via `document.querySelector("input")` and the other by `document.form[0].input`. In such a case, it is often important to determine whether the two variables point to the same

object (i.e., point-to analysis). However, it can be considerably hard to determine this by looking at the source code, as DOM trees are often generated by the same program.

(C5) Shared Third-party Code. Most modern web applications include at least one third-party JavaScript library [59], such as jQuery [12], to benefit from their powerful abstractions over the low-level browser APIs. Detection of client-side CSRF requires the ability to determine when the program performs HTTP requests, also when the program delegates low-level network operations to libraries. Similarly, library functions can be part of the data flows of a program.

To date, existing approaches are highly inefficient as they include the source code of libraries in the analysis. We observe that external libraries account for 60.55% of the total JavaScript lines of code of each web page[2], thus requiring existing techniques to re-process the same code even when visiting a new page of the same web application. An alternative approach consists of creating hand-crafted models of libraries (see, e.g., [48]). While such an approach is effective when modeling low-level browser APIs, it does not scale well to external libraries. First, external libraries are updated more frequently than browser APIs and second, there are many alternative libraries that a JavaScript program can use [31].

2.3 Overview of our Approach

To overcome our challenges, we propose *hybrid property graphs* (hereafter HPGs), a canonical, graph-based model for JavaScript programs. Also, we propose JAW, a framework that constructs HPGs starting from a seed URL, and detects client-side CSRF leveraging declarative graph traversals.

Addressing challenges. Our approach addresses our challenges as follows:

(C1) HPGs provide a uniform canonical representation for JavaScript source code, similarly as code property graphs for C/C++ [91] and PHP [33].

(C2) We define HPGs and develop JAW to enable us to perform a variety of security tasks, i.e., detection and exploratory analyses of the client-side CSRF vulnerability. We believe that decoupling the code representation (the graph) from the analysis (traversals) potentially renders JAW more suitable for reuse (like other CPG-based approaches [33, 91]). In this paper, however, we do not target nor claim the HPG reusability, as our objective is to study client-side CSRF.

(C3) HPGs captures JavaScript nuances such as event-based transfer of control by proposing the Event Registration, Dispatch and Dependency Graph (ERDDG).

(C4) HPGs captures the dynamics of the web execution environment of client-side JavaScript programs via both snapshots of the web environment (e.g., DOM trees) and traces of JavaScript events.

[2]We calculated the fraction of library lines of code over the testbed web applications of §5.1 using the crawler and the configuration of the data collection phase of §4.1.

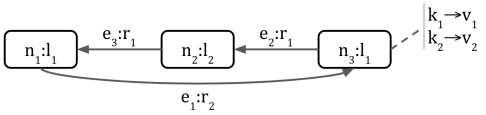
(C5) JAW can generate reusable symbolic models of external libraries, that will be used as proxy in our HPGs.

Overview. JAW takes in input a seed URL of the application under test. Then, it uses a web crawler to visit the target. During the visit, JAW stores the JavaScript and HTML code, and monitors the execution capturing snapshots of the DOM tree, HTTP requests, registered handlers, and fired events. By using a database of known signatures for common libraries, JAW identifies external libraries and generates a symbolic model for each of them. The symbolic model consists of a mapping between elements of the library (e.g., function names) and a set of semantic types characterizing their behaviors. Then, JAW builds the HPG for each stored page, and link the HPG with the pre-generated semantic models. Finally, JAW can query the HPG for detection or interactive exploration of client-side CSRF vulnerabilities.

3 Hybrid Property Graph

This section introduces hybrid property graphs (HPGs). A HPG comprises of the *code representation* and *state values*. The code representation unifies multiple representations of a JavaScript program whereas the state values are a collection of concrete values observed during the execution of the program. We use a *labeled property graph* to model both, in which nodes and edges can have labels and a set of key-value properties. The example below shows a graph where l_i is the node label and r_j is the relationship label. Nodes and edges can store data by using properties, a key-value map.

Figure 3: Example of labeled property graph



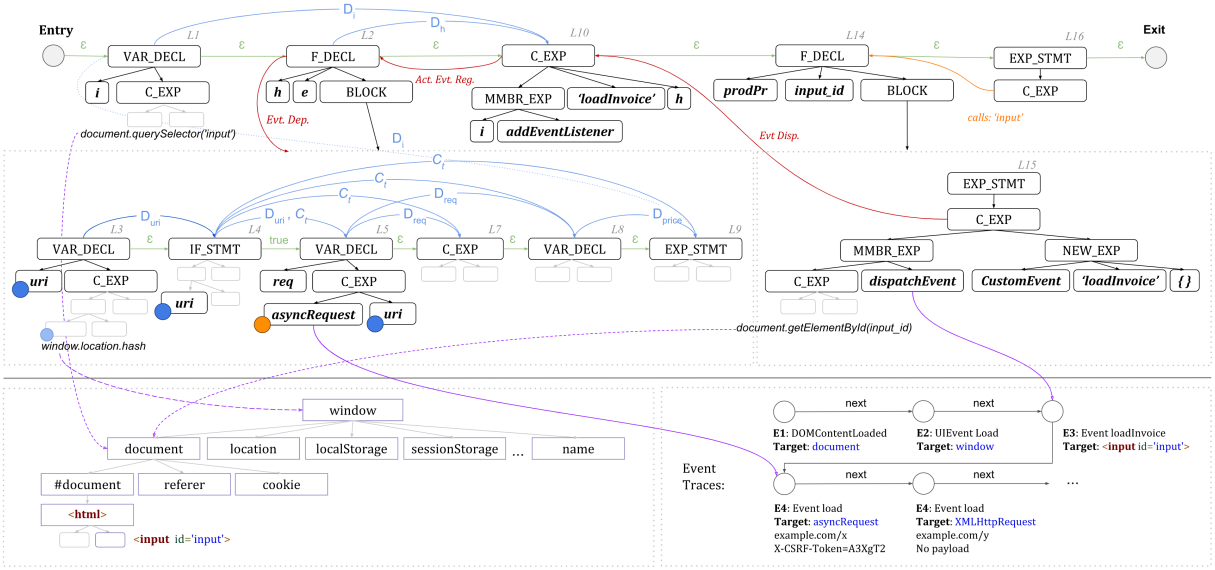
In the rest of this section, we present how we map the code representation and state values into a graph (Sections 3.1 and 3.2), and show how we can instantiate and query such a graph to study client-side CSRF vulnerabilities (§3.3).

3.1 Code Representation

The code representation models the JavaScript source code and builds on the concept of code property graph (CPG) which combines three representations for C programs, i.e., abstract syntax tree, control flow graph, and program dependence graph [91]. Later, the same idea has been adapted to study PHP programs [33], extending CPGs with call graphs. HPGs further extend CPGs with the event registration, dispatch, and dependency graph and the semantic types.

Abstract Syntax Tree (AST). An AST is an ordered tree encoding the hierarchical decomposition of a program to its syntactical constructs. In an AST, terminal nodes represent operands (e.g., identifiers), and non-terminal nodes correspond to operators (e.g., assignments). In Figure 2, AST nodes are represented with rounded boxes. Terminal nodes are in bold-italic, whereas non-terminal nodes are all capitals. AST

Figure 2: HPG for the running example in Listing 1. The top part depicts the code representation, including the AST (black edges), CFG (green edges), IPCG (orange edges), PDG (blue edges), ERDDG (red edges), and the semantic types (blue and orange filled circles representing `WIN.LOC` and `REQ` types, respectively). Note that not all nodes and edges are shown for brevity. Edges connected to dotted boxes reflect that the edge is connected to each node within the box. The bottom part demonstrates the dynamic state values to augment the static model. Arrows between the two parts represent the link between the two models.



edges connect AST nodes to each other following the production rules of the grammar of the language, e.g., in line 10 of Listing 1, `i.addEventListener('loadInvoice', h)` is a call expression (CALL_EXP) with three children, the member expression (MMBR_EXP) `i.addEventListener`, the literal `'loadInvoice'` and an identifier `h`. AST nodes are core nodes of the code representation, providing the building blocks for the rest of the presented models.

Control Flow Graph (CFG). A CFG describes the order in which program instructions are executed and the conditions required to transfer the flow of control to a particular path of execution. In Figure 2, CFG is modeled with edges (in green) between non-terminal AST nodes. There are two types of CFG edges: conditional (from predicates and labeled with *true* or *false*) and unconditional (labeled with ε). A CFG of a function starts with a *entry* node and ends with a *exit* node, marking the boundaries of the function scope. These fragmented intra-procedural flows are connected to each other by inter-procedural *call* edges, as discussed next.

Inter-Procedural Call Graph (IPCG). An IPCG allows inter-procedural static analysis of JavaScript programs. It associates with each call site in a program the set of functions that may be invoked from that site. For example, the expression `showInvoicePrice('input')` of line 16 in Listing 1 calls for the execution of the function `showInvoicePrice` of line 14. We integrate the IPCG in our code representation with directed *call* edges, e.g., see the orange edge between the C_EXP AST node and the F_DECL AST node in Figure 2.

Program Dependence Graph (PDG). The value of a variable depends on a series of statements and predicates, and a PDG [41] models these dependencies. The nodes of a PDG are non-terminal AST nodes, and edges denote a *data*, or *control* dependency. A data dependency edge specifies that a variable, say x, *defined* at the source node is afterwards *used* at the destination node, labeled with D_x. For example, in Figure 2, variable `uri` is declared in line 3 (by VAR_DECL), and used in line 4 (in IF_STMT), and thus a PDG edge (in blue) connects them together. A control dependency edge reflects that the execution of the destination statement depends on a predicate, and is labeled by C_t, or C_f corresponding to the true, or false condition, e.g., the execution of the CALL_EXP in line 7 depends on the IF_STMT predicate in line 4.

Event Registration, Dispatch and Dependency Graph (ERDDG). The ERDDG intends to model the event-driven execution paradigm of JavaScript programs and the subtle dependencies between event handlers. In an ERDDG, nodes are non-terminal AST nodes, and we model execution and dependencies with three types of edges. The first edge models the registration of an event, e.g., line 10 in Listing 1 registers `h` as the handler for the custom event `loadInvoice`. We represent the registration of an event with an edge of type *registration* between the node C_EXP (i.e., the call site for `addEventListener`) and the node F_DECL (i.e., the statement where the function `h` is defined). The second edge models the dispatch of events. For example, line 15 in Listing 1 calls the browser API `dispatchEvent` to schedule the execution of the handler of the `loadInvoice` event type. We model the transfer of control with an edge of type *dispatch*. See, for example, the edge (in red) between the C_EXP node of line 15 and the C_EXP registering the handler in Figure 2. The last edge models dependencies between statements and events. We implement the dependency with an edge between the AST

node for the handler's declaration and the AST nodes of the handler's statements. Figure 2 shows such an edge from the F_DECL node of line 2 and the body of the function.

Semantic Types. The detection of client-side CSRF requires identifying statements that send HTTP requests, and that consume data values from pre-defined sources. We model the properties of statements via semantic types. A semantic type is a pre-defined string assigned to program elements. Then, types are propagated throughout the code, following the calculation of a program, e.g., we can assign the type `WIN.LOC` to `window.location` and propagate it to other nodes, following PDG, CFG, IPCG, and ERDDG edges. In Figure 2, we use a blue filled circle for the type `WIN.LOC` that is propagated following the D_{uri} PDG edge, i.e., the term *uri* of line 3, 4, and 5. Semantic types can also be assigned to functions to specify their behavior abstractly. For example, we can use the string `REQ` for all browser APIs that allow JavaScript programs to send HTTP requests, such as `fetch`, or `XMLHttpRequest`. HPGs model semantic types as properties of the AST node.

Symbolic Modeling. When analyzing the source code of a program, we need to take into account the behaviors of third-party libraries. We extract a symbolic model from each library and use it as a proxy for the analysis of the application code. In this work, the symbolic model is an assignment of semantic types to libraries' functions and object properties. For example, in Figure 2, we can use the semantic type `REQ` (represented with an orange filled circle) for the `asyncRequest` term, and abstract away its actual code. Also, to reconstruct the data flow of programs that use library functions, we define two semantic types modeling intra-procedural input-output dependencies of library functions. We use the semantic type $o \leftarrow i$ for functions whose input data values flow to the return value and the type $o \sim i$ for functions whose output is conditioned on the input value (e.g., by an IF_STMT). As we will show in §4, the symbolic modeling of libraries is performed automatically by JAW, who creates a mapping between the library elements and a list of semantic types.

3.2 State Values

JavaScript programs feature dynamic behaviors that are challenging to analyze via static analysis. As such, we augment HPGs to include concrete data values collected at run-time, and link them to the counterpart code representation.

Event Traces. To capture the possible set of fired events that are not modeled due to the limitations of the static analysis [46], or auto-triggered events, we augment the static model with dynamic traces of events. Event traces are a sequence of concrete incidents observed during the execution of a web page. For example, the `load` event or a network event for the response of a HTTP request. We use the trace of events fired upon the page load to activate additional *registration* edges in our ERDDG graph when possible. As shown in Figure 2, the nodes of the graph for event traces represent concrete events observed at run-time, and edges denote their ordering.

Figure 4: Examples of vulnerable code. Orange and blue boxes represent `REQ` and `WIN.LOC` semantic types, respectively.

```
1
1. var domain = validate_domain(window.location.hash);
2. fetch("https://" + domain + "/"); <-

2
1. var uri = window.location.hash;
2. var xhr = new XMLHttpRequest();
3. xhr.open("GET", uri); <-
4. xhr.send();

3
1. var uri = validate_uri(window.location.hash);
2. var xhr = new extLibraryHttpRequest();
3. let a = xhr; // alias
4. a.open("POST", uri); <-
5. var q = window.location.search;
6. a.send("q1="+q.substr(1,10)+ "&q2="+q.substr(11); <-
```

Environment Properties. Environment properties are attributes of the global `window` and `document` objects. The execution path of a JavaScript program and the values of variables may differ based on the values of the environment properties. We enrich HPGs by creating a graph of concrete values for the properties observed dynamically. We also store a snapshot of the HTML DOM tree [65]. If the value of a variable is obtained from a DOM API, the actual value can be resolved from the tree. We use the DOM tree to locate the objects that a DOM API is referencing. For example, to determine if an event dispatch is targeting a handler, we can check if the dispatch and registration is done on the same DOM object. We create a node for each environment property, and store concrete values as properties of the node. As depicted in Figure 2, we connect these nodes by edges representing a property ownership, or a parent-child relationship.

3.3 Analysis of Client-side CSRF with HPGs

Given a HPG as described in Sections 3.1 and 3.2, we now use it to detect and study client-side CSRF. We say that a JavaScript program is vulnerable to client-side CSRF when (i) there is a data flow from an attacker-controlled input to a parameter of an outgoing HTTP request *req*, and (ii) *req* is submitted on the page load.

We model both conditions using graph traversals, i.e., queries to retrieve information from HPGs. In our work, we define graph traversals using the declarative Cypher query language [3], but in this paper we exemplify Cypher syntax with set notation and predicate logic while retaining the declarative approach. A query Q contains all nodes n of HPG for which a predicate p (i.e., a graph pattern) is true, i.e., $Q = \{n : p(n)\}$. We use predicates to define a property of a node. For example, we use the predicate $hasChild(n, c)$ to say that a node n has an AST child c. Another example of predicate is $hasSemType(n, t)$, which denotes a node n with a semantic type t. Predicates can be combined to define more complex queries, e.g., via logical operators.

Detection of Client-side CSRF. The first condition for client-side CSRF vulnerability is the presence of attacker-controlled input parameters for outgoing requests. Figure 4 shows different instances of vulnerable code taken from real examples, where by construction, we assigned the `WIN.LOC` and `REQ` semantic types to AST nodes, which are shown as

blue and orange boxes, respectively. For all three cases of Figure 4, the goal is to identify the lines of code having both orange and blue labels (marked with a red arrow). At a high level, a line of code is a non-terminal AST node for JavaScript statements or declarations (e.g., EXP_STMT, VAR_DECL), that we represent with the predicate $isDeclOrStmt(n)$. Then, once we identify such an AST node n, we need to explore whether the node has two children c_1 and c_2 where one is of type `REQ` and the other is of type `WIN.LOC`. Following our notation for queries, we can write:

$$N_1 = \{n : isDeclOrStmt(n) \wedge \exists c_1, c_2,\ c_1 \neq c_2 \wedge hasChild(n,\ c_1) \wedge hasSemType(c_1,\ \text{“}REQ\text{”}),\ \wedge\ hasChild(n,\ c_2) \wedge hasSemType(c_2,\ \text{“}WIN.LOC\text{”})\} \quad (1)$$

Query 1 is not a sufficient condition to determine the presence of a client-side CSRF vulnerability, as the returned nodes may correspond to lines of code not executed at page load. We refine it with additional checks for reachability. In general, starting from a node n such that $isDeclOrStmt(n)$, we could follow backward CFG edges (both ε, $true$, and $false$) to determine whether we reach the CFG entry node. Then, whenever we reach a function definition (e.g., F_DECL), we jump to all its call sites following the IPCG call edges. But this will not be sufficient because a function can be executed when a specific event is fired. Accordingly, we need to visit backward the ERDDG edges i.e., the dependency edge, followed by the registration and the dispatch edge. We handle separately special cases where events are fired by the browsers automatically during loading a page. We keep on following backward CFG, ERDDG, and IPCG edges until either we reach the CFG entry node or when there are no longer nodes matching any of the previous criteria. We say that a node n is reachable if the CFG entry node is in the query result set.

Analysis of Vulnerable Behaviors. The previous queries can identify the general vulnerable behavior of client-side CSRF, i.e., a program that submits a HTTP request using attacker-chosen data values. However, programs may implement a variety of checks on the inputs, which can eventually influence the exploitation landscape. In Figure 4, for example, Program 1 shows a vulnerable script whose domain validation of line 1 restrains the attacker from manipulating the entire request URL. Program 2, however, shows a case where the attacker can chose the complete URL string, including the path and query string. One aspect of client-side CSRF vulnerabilities that we intend to study is to identify the extent to which an attacker can manipulate the outgoing request. For instance, if `window.location` properties flow to a request parameter without any sanitization. Query 2 captures this aspect:

$$N_2 = \{n_1 : \forall n_1 \in N_1,\ \exists n_2,\ hasPDGPath(n_2,\ n_1) \wedge isAssignment(n_2) \wedge \exists c,\ hasChildOnRight(n_2,\ c) \wedge isMemberExp(c) \wedge hasValue(c, \text{“}window.location\text{”})\} \quad (2)$$

Query 2 checks if the node n_1 returned by Query 1 is connected via PDG edges to an assignment statement whose right-hand side child is a property of the `window.location`. The predicate $hasPDGPath(n_2,\ n_1)$ specifies that there is a path from n_2 to n_1 following PDG edges, and $isAssignment(n_2)$ marks that n_2 is a VAR_DECL, or an ASSIGN_EXP node.

Another aspect to consider is the number of attacker-controllable items within a request. For example, Program 3 of Figure 4 shows a more complex example where the attacker can also control the content of the request body, increasing the flexibility to create an exploit for the vulnerable behavior. For this, a query can cluster vulnerable lines of code that belong to the same HTTP request, making use of the PDG dependencies among elements of the same request. Then, the query can count the number of attacker-controllable injection points (see, e.g., the two injection points in line 6 of Program 3 as well as the injection point in line 4).

4 JAW

In this section, we present JAW, a framework to study client-side CSRF vulnerabilities using HPGs. Starting from a seed URL of a web site, JAW visits web pages using a JavaScript-enabled web crawler to collect the web resources. During the visit, JAW also collects run-time state values. Then, given a list of user-defined semantic types and their mapping to JavaScript language tokens, JAW constructs the HPG. The construction has two phases. First, JAW identifies external JavaScript used by the program and processes it in isolation to extract a symbolic model. Then, it constructs the graph of the rest of the JavaScript code, and link elements of the JavaScript program to the state values. Finally, JAW analyzes client-side CSRF by executing queries on the HPG (§3.3). Figure 5 shows an overview of the JAW's architecture.

4.1 Data Collection

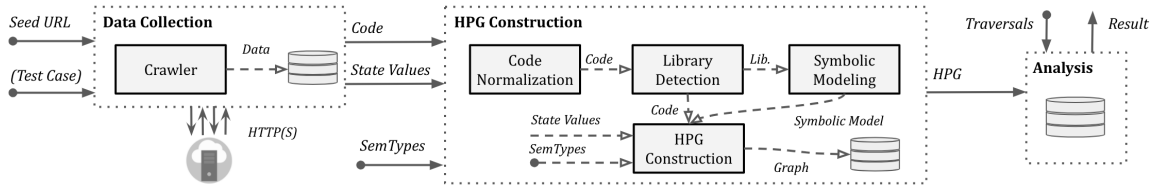
The data collection module performs two tasks: crawling to discover URLs from different user states, and collecting the JavaScript code and state values for each web page found.

Input. The input of the data collection module is a seed URL of the web application under test, and, optionally, test cases to pass the user login, e.g., as scripted Selenium tasks [17] or via trace recording [15, 16].

Crawler. We developed a crawler that uses a headless instance of Chrome [10] controlled via Selenium [17]. Starting from the seed URL, the crawler visits the web application to collect web resources and run-time execution data. It follows the iterative deepening depth-first search strategy, and terminates when no other URLs are found, or when its allocated time budget runs out (default is 24h). Optionally, if provided as input, it executes test cases before the crawling session.

JavaScript Code and State Values. When visiting each page, the crawler stores the web resources and state values every $t_i = 10$ seconds for $m = 2$ times (configurable parameters). The crawler collects the HTML page, JavaScript program, fired events, HTTP requests and responses, and the JavaScript properties explicitly shown in (bottom left of) Figure 2 for each t_i interval. While JavaScript properties are extracted via

Figure 5: Architecture of JAW.



the Selenium interface, we developed a Chrome extension for our crawler that resorts to function hooking to intercept calls to the `addEventListener` for collecting events and to the `chrome.webRequest` API to intercept the network traffic.

4.2 Graph Construction

JavaScript code and state values collected are next used to build a HPG. The built graph is imported into a Neo4j [14] database allowing for fine-grained, declarative path traversals to detect and study client-side CSRF. This section delineates technical details for constructing HPGs.

Normalizing JavaScript Code. As a first step, JAW creates a normalized JavaScript program by concatenating code segments inside the script tags and HTML attributes (i.e., *inline* JavaScript code), preserving the execution order of program segments. When combining inline code, JAW replaces inline event handler registration with `addEventListener` API.

Library Detection. To identify libraries, we use Library Detector [13], a tool that searches for known library signatures inside the execution environment (e.g., global variables)[3].

HPG Construction. JAW constructs HPGs as follows. First, a graph is created for the symbolic modeling of each detected library. This step is skipped if a symbolic model for the library already exists. Then, it creates a graph for the program under analysis. Regardless the use of the graph, the rules to construct a HPG do not change, as presented next.

1. AST—JAW uses Esprima [7], a standard-compliant ECMAScript [11] parser to generate the AST of the normalized source code. The output of Esprima is a JSON representation of the AST. In this representation, a node is a key-value dictionary with a `type` property (e.g., VAR_DECL) and edges are represented with ad-hoc dictionary keys. We mapped the JSON output to AST nodes and AST edges of our graph.

2. CFG— We extensively reviewed open-source CFG generators, such as escontrol [5], styx [18], or ast-flow-graph [1], and selected Esgraph [6] because of its popularity, and compliance with Esprima. Starting from an AST, Esgraph generates a CFG where nodes are AST nodes for statements or declarations, and an edge is labeled with `true` or `false`, for a conditional branch, or ε for a node of the same basic block.

3. PDG—JAW uses dujs [4], a def-use analysis library based on Esgraph. We modified dujs to add support for global variables, closures, and anonymous function calls. The output of dujs is a list of def-use relationships for each variable v between the AST edges, that JAW import as data dependence edges D_v in our HPG. For the control dependence edges, JAW calculates post-dominator trees [58] from the CFG, one for each statement s. Then, JAW maps each edge of the tree to C_t or C_f for the true or false branch, respectively.

4. IPCG—JAW generates the IPCG as follows. During the construction of the AST and CFG, JAW keeps track of all function definitions and call sites. Then, it associates a call site to the function definition(s) it may invoke. There are five types of call expressions in JavaScript: function calls on the global object (e.g., `foo()`), property calls (e.g., `a.foo()`, or `a['foo']()`), constructor calls (e.g., `new Foo()`), invocations via the `call()` [9], and `apply()` method [8]. For all cases, the actual function definition name may be aliased. We resolve the pointers using our PDG, and connect the call edge accordingly. If the value of the pointer is conditioned, we connect an edge to each respective function definition.

5. ERDDG—For the generation of the ERDDG, JAW keeps track of event dispatches and handler registrations during the creation of the AST and the CFG. For each event handler found, JAW creates a registration edge that connects the top-level AST node (i.e., CFG node) to the handler function, and a dependency edge connecting the handler function to statements of the body. To associate each event dispatch to a registration site, we check if they target the same DOM element. For this, we resolve the pointer on which the event is dispatched, and the pointer on which the handler is registered leveraging our PDG, and check if they refer to the same variable declaration or different variables with verbatim or semantically same values. We use the DOM snapshot to check if two different DOM queries can semantically target the same element. For example, an element can be queried with its *id*, or alternatively its *name* attribute. Once we determine that the pointers reference the same element, we connect an edge between the dispatch and registration sites.

6. Semantic Types and Propagation— The input for this step is a mapping T between a semantic type t and a signature for AST node σ, e.g., we map the type `WIN.LOC` to the JavaScript property `window.location`. For each pair $(t,\sigma) \in T$, JAW stores each type t to the AST node that is matching the signature σ. Then, JAW propagates the type t through the HPG.

Algorithm 1 propagates forward a type t from a node n to other nodes. First, the function `propagateLeft` assigns the type t to the variable v on the left-hand side (e.g., of an assignment), if any, and returns it. Then, the function

[3] We refer interested readers to Appendix A.2.

Algorithm 1: Forward semantic type propagation

```
inputs  : Node n with a variable having semantic type t.
outputs : Propagates semantic types and returns the last tainted node.
1  function propagateForward(n, t):
2      v ← propagateLeft(n, t) // taint left-hand side
3      n_t ← n // last tainted node
4      P ← propagateByPDG(n, v, t) // tainted PDG paths
5      for p_i ∈ P do
6          n_t ← p_i[p_i.length − 1] // last CFG-level tainted node
7          v_t ← getRightHandSideTaintedVariable(n_t, t)
8          if hasSymbolicFunctionCall(n_t) and hasSemanticType(n_t,
             "o<-i") then
9              o ← propagateLeft(n_t, t)
10             propagateForward(o, t) // recursion
11         end
12         if hasCallExpressionWithCallArgOfType(n_t, t) then
13             c ← traverseCallEdge(n_t, v_t, t) // call def param
14             ret ← propagateForward(c, t) // returned variable
15             if isRetStmt(ret) and hasSemanticType(ret, t) then
16                 v_left ← propagateLeft(n_t, t)
17                 if v_left is not null then
18                     propagateForward(v_left, t) // recursion
19                 end
20             end
21         end
22         if hasDispatchEdgeWithArgOfType(n_t, t) then
23             e ← traverseDispatchAndRegistrationEdges(n_t, v_t, t) //
                 handler param
24             propagateForward(e, t)
25         end
26     end
27     return n_t // last tainted node
```

`propagateByPDG` propagates t following PDG edges and returns the visited paths P. Then, for each node n_t at the end of the path $p_i \in P$, we distinguish three cases. The first case is that n_t is a function call that is modeled by the special semantic types assigned during the symbolic modeling. If so, we taint the output variable o, and recursively call `propagateForward` for o. Second, n_t is a call expression having an IPCG edge. In this case, we taint the parameter c on the function definition corresponding to the argument tainted on the call site, and call `propagateForward` for c. Then, we check if the last tainted node from the context of the function definition is a tainted return statement. If so, we call `propagateForward` for the variable v_{left} on the call site that holds the returned result. Third, n_t is an event dispatch expression that passes tainted data. In this case, we jump the dispatch and registration edges, taint the corresponding event variable, and call `propagateForward` for the variable. This process terminates when none of the above criteria holds.

JAW performs the semantic type propagation when creating both the HPG for the symbolic modeling of a library and the HPG of the rest of the code. When creating the HPG for the rest of the code, the semantic type mapping T includes the mapping created during the symbolic modeling.

Symbolic Modeling. The output of this step is a mapping of semantic types and AST nodes, which is used during the construction of a HPG for the program under analysis. Symbolic modeling starts with the construction of a HPG from the library source code. Then, after the propagation of the semantic types, JAW searches for function definitions with intra-procedural input-output relationships. More specifically, JAW identifies all non-anonymous function expressions with at least one input parameter, and track the value of its return statement(s), if any, through a backward program slicing approach. At a high level, we start from where a value is returned, flow through where it is modified, and end at where it is generated leveraging the PDG, CFG, IPCG, and ERDDG graphs. If the returned variable, say o, has a PDG *control* dependency to a function input, say i, we assign the type $o \sim i$ to the function. If we establish a PDG *data* dependency, we mark it with $o \leftarrow i$. Finally, JAW selects all function expression and object property nodes with at least one semantic type, that will be used in the HPG construction of the JavaScript code.

5 Evaluation

The overarching goal of our evaluation is to study client-side CSRF vulnerabilities and to assess the efficacy and practicality of JAW. We run JAW on 4,836 web pages, crawled from 106 popular web applications, generating HPGs for 228,763,028 LoC. During this process, we discover 12,701 forgeable client-side requests split across 87 applications. We find that seven applications suffer from at least one zero-day client-side CSRF vulnerability that can be exploited to perform state-changing actions and violate the server's integrity.

Before presenting the evaluation results, we discuss the experimental setup (§5.1) and show properties of problem space and how JAW tackled them (§5.2). Then, we report the findings of our experiments (§5.3), and finally, conclude with the analysis of JAW's results (§5.4).

5.1 Experimental Setup and Methodology

Testbed. We select web applications from the Bitnami catalog [2] that offers ready-to-deploy containers of preconfigured web applications. We choose Bitnami applications due to their popularity (e.g., see [19]), diversity, and use by prior work (e.g., see [69]). At the time of the evaluation, Bitnami contains 211 containers. We discard 105 containers without web applications and duplicates, e.g., the same web application using different web servers. The remaining 106 web applications are: 23 content management system, 15 analytics, 11 customer relationship management, ten developer tools and bug tracking, eight e-commerce, eight forum and chat, five email and marketing automation, four e-learning, three media sharing, two project management, two accounting and poll management, and 15 other. The complete list of web applications is in Appendix B.1, among which we have WordPress, Drupal, GitLab, phpMyAdmin, and ownCloud.

Then, for each web application, we created one user account for each supported levels of privilege and a Selenium test case to perform the login. In total, we created 136 test scripts, ranging from one to five test cases per application.

JAW Inputs. The inputs of JAW are the seed URLs, the Selenium test cases, and a semantic type mapping. The seed URLs contain the URLs for the user login (total of 113 login URLs), whereas the test cases are the ones we prepared when configuring the testbed. Then, for all web applications, we used the semantic types listed in Table 4 in Appendix A.1.

Methodology for Client-side CSRF Detection. We de-

ployed the web applications under evaluation locally, and instantiated JAW against each of the targets. After the data collection and creation of the HPGs, we run a set of queries to identify attacker-controllable requests. We then use additional queries to identify the request fields under the control of the attacker and the type of control. We assess the accuracy of the query results via manual verification. For each forgeable request, we load the page in an instrumented browser and verify whether the manipulated inputs are observed in the client-side requests. For example, if the request uses data values of type `WIN.LOC`, we inject a token in the vulnerable page URL and search the token in the outgoing request. After confirming the forgeability of the request, we look for its use in an attack. First, we search for server-side endpoints performing security-relevant state-changing actions, such as modifying data on the server-side storage. Then, we construct a string that, when processed by the vulnerable page, it will result in a request towards the identified endpoint. Finally, we pack the string into a malicious URL and verify whether the attack works against a web application user with a valid session, who clicks on the URL.

Methodology for Impact of Dynamic Snapshotting. We performed additional experiments to assess the impact of our dynamic snapshotting approach in (i) vulnerability detection, and (ii) HPG construction. First, we prepared a variant of JAW, hereafter referred to as JAW-static, which follows a pure static approach for HPG construction and analysis (§3.1). Specifically, JAW-static does not consider the following dynamic information: fired events, handler registrations, HTTP messages, global object states, points-to analysis for DOM queries, dynamic insertion of script tags, and the DOM tree snapshot. We repeated our evaluation using JAW-static, and determined the lower bound of false negative and false positive vulnerabilities in JAW-static by comparing it to JAW's evaluation results. Also, we compare the differences in HPG nodes, edges and properties.

Then, we logged all the fired events that are not auto-triggered and that JAW failed to find their line of code for HPG construction. Such cases are an indication of false negative edges in HPGs generated by JAW. Accordingly, we manually review all cases to uncover the reasons for false negative edges. Finally, we conducted another experiment to assess the false positive and false negative edges as a result of using the DOM tree snapshots for points-to analysis of DOM queries. For all web pages, we instrumented the JavaScript code to log the actual element a DOM query is referring to, and compared it against the value that JAW resolved. JAW uses these resolutions to create ERDDG edges, opening the possibility for both false positive and false negative edges.

5.2 Analysis of Collected Data

Size of the Analysis. Starting from 113 seed URLs, JAW extracted 4,836 web pages, ranging from 1 to 456 web pages per web application, and about 46 web pages per application. The structural analysis of these URLs reveals that 865 of them have a hash fragment, an indication that these URLs carry state information for the client-side JavaScript program—a characteristic of single-page web applications. In total, 39 web applications use URLs with hash fragments.

From the 4,836 pages, JAW extracted 228,763,028 LoC, which amounts to generating 4,836 HPGs by processing about 47,304 LoC per page. When looking at the origin of the code, we observed that the majority of it, i.e., 60.55%, is from shared libraries, e.g., jQuery (28,645 LoC per page and 138,525,092 LoC in total), whereas the remaining is application code in script tags (39.42% or 18,649 LoC per page, over 90,188,256 LoC in total) and a negligible amount is inline code (0.02% or 10 LoC per page, over 49,680 LoC in total).

Finally, at run-time, we observe that about 2.63% of the script tags are loaded dynamically (i.e., by inserting a script tag programmatically), over a total of 104,720 script tags. Also, JAW observed 51,974 events that are fired when loading the page (about 11 events per page) distributed across 46 event types, of which 38 are HTML5 types (e.g., animation and DOM mutation events) and 8 are custom. As we will show next, even if the number of run-time monitored events is negligible, their role in the analysis is fundamental.

Importance of Symbolic Modeling. The analysis of client-side programs requires to process 228,763,028 LoC of which 138,525,092 of them are for the libraries alone, about 60% of the total. Our analysis reveals that libraries are largely reused both across web applications and across pages. First, the 106 web applications in our testbed use in total 31 distinct libraries. Second, each page contains from zero to seven script libraries, with an average number of two libraries per page. Third, the total amount of code of the 31 libraries is 412,575 LoC, which is 335 times smaller than the total 138,525,092 LoC across all pages. Accordingly, pre-processing the library code to extract a symbolic model reduces by more than half (-60.37%) the effort required to generate HPGs, moving from 228,763,028 LoC to 90,650,511 (i.e., the sum of LoCs of the application, inline JavaScript, and the 31 libraries).

For each of the 31 libraries, JAW generates one HPG and extract a symbolic model. Table 1 shows an overview of the results of the symbolic modeling step. In total, JAW modeled 11,977 functions in around half an hour, half of which have the input-output relationship semantic types (i.e., 5,923 functions)—a relevant function behavior to correctly reconstruct the data flows of a program.

Role of ERDDG. In total, JAW generated 4,836 HPGs, one for each page, for a total of 508,810,412 nodes and 652,662,573 edges. Of these edges, the ones that are crucial to analyze JavaScript programs are those modeling the transfer of control via event handlers. In total, JAW identified 64,854,097 event edges (i.e., registration, dependence and dispatch) of which 6,451,582 are dispatch edges, i.e., edges modeling the intention to execute the event handlers. For comparison, the number of call edges that also transfer the control

Library	Usage %	LoC	Funcs.	*I/O*	Time (s)
JQuery	81.13%	10,872	428	238	57.54
Bootstrap	38.67%	2,377	55	55	41.07
JQuery UI	27.35%	18,706	320	320	82.33
ReactJS	9.43%	3,318	130	40	39.59
ReactDOM	9.43%	25,148	688	368	81.98
RequireJS	8.49%	1,232	50	50	35.72
AngularJS	5.66%	36,431	852	558	82.92
Analytics	5.66%	20,345	244	233	69.21
Backbone	5.66%	2,096	148	50	38.26
Modernizer	5.66%	834	292	21	34.50
Prototype	5.66%	7,764	266	243	54.10
YUI	4.71%	29,168	2,414	637	149.34
JIT	3.77%	17,163	430	255	69.11
ChartJS	2.83%	16,152	263	253	76,75
Dojo	2.83%	18,937	696	313	63.32
LeafletJS	2.83%	14,080	650	208	62.65
Scriptaculous	2.83%	3,588	97	84	46.11
HammerJS	1.88%	2,643	67	47	37.01
MomentJS	1.88%	4,602	138	138	45.44
ExtJS	1.88%	135,630	2,701	1,135	231.86
Vue	1.88%	11,965	638	288	62.77
YUI History	1.88%	789	20	10	18.41
Bootstrap Growl	0.94%	215	7	7	32.21
Bpmn-Modeler	0.94%	19,139	231	228	65.84
CookiesJS	0.94%	79	3	0	31.29
FlotChartsJS	0.94%	1,267	15	15	42.38
GWT WebStarterKit	0.94%	110	3	2	31.15
Gzip-JS	0.94%	280	4	4	31.87
Handlebars	0.94%	6,726	103	103	50.83
SpinJS	0.94%	190	4	4	31.99
SWFObject	0.94%	729	20	16	33.61
Total		412,575	11,977	5,923	1919.84

Table 1: Symbolic modeling of shared JavaScript libraries.

to other sites of a program, are 7,179,021, meaning that the ERDDG representation enables the identification of +89.87% edges transferring the program control.

5.3 Forgeable Requests

The first step to detect client-side CSRF is the identification of lines of code that can generate attacker-controlled requests. For that, we prepared a set of queries as discussed in §3.3. Based on our threat model (§2.1), we considered different attacker-controlled inputs for JavaScript programs (see [60]) that can be forged by different attackers.

JAW identified 49,366 lines of code across 106 applications that can send an HTTP request, and marked 36,665 of them as unreachable during the page load or not using attacker-controlled inputs. The remaining 12,701 requests could be controlled by an attacker. We grouped these requests by the semantic types of the input source corresponding to different attackers (see §2.1), as shown in Table 2. We observe that the majority of applications, i.e., 87, sends at least one forgeable request at page load.

False Positives. Considering the high number of forgeable requests, we could not verify all of them via manual inspection. Instead, we first selected all requests across all groups, except for `DOM.READ`. Then, for `DOM.READ`, we focused on one request (randomly selected) for each web application, i.e., 83 requests. In total, we inspected 516 forgeable requests. For the inspection, we loaded the vulnerable page in an instrumented browser to inject manipulated strings and observe whether the outgoing requests include manipulated strings. We confirmed that all requests, except for one of the 83 `DOM.READ` requests include the manipulated content. After a careful investigation, we observed that the false positive occurs as a result of inaccurate pointer analysis of the context-sensitive `this` keyword, which has a run-time binding, and may be different for each invocation of a function depending on how the function is called, e.g., dynamically called functions, or different invocation parameters using a hierarchy of `call` and `apply` methods [8, 9] lead to different bindings of `this`.

Sources	Forgeable	Apps
`DOM.COOKIES`	67	5
`DOM.READ`	12,268	83
`*-STORAGE`	76	8
`DOC.REFERRER`	1	1
`POST-MESSAGE`	8	8
`WIN.NAME`	1	1
`WIN.LOC`	280	12
Total forgeable	12,701	87
Non-reachable code	36,665	101
Total	49,366	106

Table 2: Number of forgeable requests and affected web applications.

Exploitations. Next, we looked for practical exploitations for the 515 requests manually. In these experiments, we assumed a web attacker model for all input sources, except for cookies for which we assumed a network attacker model (see §2.1). We were able to generate a working exploit for 203 forgeable requests affecting seven web applications, all of them using data values of `WIN.LOC`, that can be forged by any web attacker. For the other groups of requests, we were not able to find an exploit. We point out that it is hard to achieve completeness when looking for exploitations manually as such a task requires extensive knowledge of web applications for identifying target URLs and the points where an attacker could inject malicious payloads. The fact that we could not find an exploit does not imply that an exploit does not exist. For these cases, we confirmed that the JavaScript code sends HTTP requests by processing data values taken from different data structures unconditionally. A highly motivated attacker could eventually find a way to inject malicious payloads in these data structures and exploit these forgeable requests.

5.4 Analysis of Forgeable Requests

In this section, we have a closer look at the degree of manipulation an attacker can have on the forgeable requests of Table 2. We extracted the stack trace for the lines of code that send forgeable requests and characterized the vulnerable behavior along three dimensions: forgeable request fields, type of manipulation, and the request template.

Forgeable Fields. First, the request field(s) that can be manipulated can determine the severity of the vulnerability. For example, if the attacker can change the domain name of a request, the client-side CSRF could be used to perform cross-origin attacks. We grouped web applications in four categories, based on the field being manipulated and found that in nine, 34, 41, and 41 web applications, an attacker can manipulate the URL domain, the URL path, the URL query string, and the body parameter, respectively. Also, we grouped appli-

Outgoing HTTP Request						Total	
Dom.	Path	Query	Body	Part	Control	Reqs	Apps
		✓		One	-, A, -	16	11
			✓	One	-, A, -	5	5
			✓	One	W, -, -	(*)166	25
			✓	One	-, -, P	1	1
	✓			One	W, -, -	28	1
	✓			One	-, A, -	7	7
	✓			One	-, -, P	6	6
		✓		One	-, -, P	11	11
	✓		✓	Mult	-, A, -	4	1
	✓		✓	Mult	W, -, -	(*)20	1
	✓	✓		Mult	W, A, P	6	1
		✓	✓	Mult	W, -, -	2	1
		✓		Mult	-, A, -	7	7
			✓	Mult	-, -, P	2	2
	✓			Mult	-, A, -	3	3
		✓		Mult	-, -, P	1	1
			✓	Mult	-, A, -	5	5
	✓			Mult	-, -, P	6	6
			✓	Mult	W, -, -	28	8
	✓	✓		Any	W, -, -	1	1
✓	✓	✓		Any	W, -, -	(*)185	8
✓	✓	✓	✓	Any	W, -, -	1	1
			✓	Any	W, -, -	(*)1	1
			✓	Any	W, -, -	2	2
	✓	✓	✓	Any	W, -, -	1	1

Legend: A=Appending; P=Prepending; W=Writing.

Table 3: Taxonomy of client-side CSRF. Each template reflects the level of attacker's control on the outgoing HTTP request. * are the templates for which we found an exploit.

cations by the number of fields that can be manipulated in a request. In total, 55, 34, and 12 applications allow modifying one, more than one, and all fields, respectively.

Operation to Forge a Field. Another factor that influences the severity is the operation that copies a manipulated value in one or more fields. We found that 28 applications allow an attacker to change the value of one or multiple fields. Also, 38 and 28 applications allow an attacker to add one or multiple fields by appending and prepending the attacker-controlled string to the final string, respectively.

Forgeable Request Templates. We characterize HTTP requests via templates, where we encode the type and number of fields that can be manipulated as well as the type of operation. Table 3 lists all templates, and for each template, it shows the number of matching requests and web applications using them. In total, we identified 25 distinct templates. We observed that the majority of web applications use only one template (i.e., 68 applications) across all their web pages or two templates (i.e., 17 applications).

5.5 Exploitations and Attacks

The 203 exploitable client-side CSRF affect seven targets, as shown next. Our exploits attack web applications the same way classical CSRFs do, i.e., by performing security-relevant state-changing requests. In addition, we found exploitations of client-side CSRF that enable XSS and SQLi attacks, which cannot be exploited via the classical attack vector.

SuiteCRM and SugarCRM. In total, we found 115 and 38 forgeable requests in SuiteCRM and SugarCRM, which can be exploited to violate the server's integrity. In both applications, the JavaScript code reads a hash fragment parameter, e.g., `ajaxUILoc`, and uses it verbatim as the endpoint to which an asynchronous request is submitted. An attacker can forge any arbitrary request towards state-changing server-side endpoints to delete accounts, contacts, cases, or tasks–just to name only a few instances that we confirmed manually.

Neos. We found eight forgeable requests in Neos. In all of them, each parameter p of the HTTP request originates from the page's URL parameter `moduleArguments[@`p`]`. Among these, we have, for example, the action and controller parameters that are used by the backend server to route the request to internal modules. Such behavior allows an attacker to direct a request to any valid internal module, including those implementing state-changing operations. For example, we exploited this behavior to delete assets from the file system.

Kibana. We found one forgeable request, generated by Timelion, a Kibana's component that combines and visualizes independent data sources. Timelion allows running queries on data sources using a own query syntax. The JavaScript code can read queries from the page's URL fragment and pass them to the server side. As a result, an attacker can execute malicious queries without the victim's consent or awareness.

Modx. We discovered 20 forgeable requests in Modx that can be exploited in two distinct ways. First, Modx's JavaScript fetches a URL string from the query parameter of the page's URL, and uses it verbatim to submit an asynchronous request with a valid anti-CSRF token. Similarly to SuiteCRM and SugarCRM, an attacker can forge requests towards state-changing server-side endpoints. Also, in one forgeable request, Modx copies a page's URL parameter in a client-side request, which is reflected back in a response and inserted into the DOM tree, allowing an attacker to use client-side CSRF to mount client-side XSS. Based on our manual evaluation, the attacker can exploit the client-side XSS only via client-side CSRF.

Odoo. We found one forgeable request that uses an `id` parameter of the URL fragment to load a database entity. We discovered that the server uses this parameter in a `SQL` query which is not properly validated, resulting in an SQLi vulnerability. We note that, due to a anti-CSRF token, the exploitation of the SQLi vulnerability via direct requests is extremely hard without exploiting first the client-side CSRF vulnerability.

Shopware. We found 20 forgeable requests sent by Shopware on page load. The code maps the page's URL hash fragment to different parts of the outgoing request. First, the code uses the first fragment of the hash fragment as URL path of the outgoing request. Then, it uses the remaining fragments as parameters of the outgoing request body. This allows an attacker, for instance, to delete products of the shop's catalog.

5.6 Impact of Dynamic Snapshotting

We designed and carried out additional experiments to show the impact of dynamic snapshotting in vulnerability detection and HPG construction (see our methodology in §5.1).

5.6.1 Vulnerability Detection

We repeated our evaluation using JAW-static, and compared the results with JAW (§5.1). In total, JAW-static found 48,543 requests, out of which 11,878 reported to be forgeable. By comparing the difference, we observed that JAW-static has detected 840 less forgeable requests (i.e., a lower bound of +7.07% false negatives). Out of the 840 false negatives, 161 cases are vulnerabilities for which we found an exploit, i.e., JAW-static does not detect 79.3% of the *exploitable* client-side CSRF vulnerabilities that was detected by JAW. Additionally, JAW-static reported 17 more cases that were not vulnerable (i.e., a lower bound of +0.15% false positives). We manually examined *all* the false positive and false negative cases to discover the underlying reasons.

False Positives (FP). Out of 17 FPs, 12 were due to non-existing dynamically fetched code (i.e., by dynamic insertion of script tags) where the value of the tainted variable changed in the dynamic code. Such FPs are eliminated in JAW because it monitors the program execution leveraging the DOM tree and HTTP messages. Then, 3 out of the 17 cases were due to a subsequent removal of the event handlers using dynamic code evaluation constructs with dynamically generated strings. Finally, the last two FPs occurred due to the removal of elements from the DOM tree, and thus the implicit removal of their event handlers. Similarly, such FPs do not occur with JAW, as it monitors the fired events and their handlers at run-time.

False Negatives (FN). We observed that almost half of the FNs, i.e., 405, occurred because the vulnerability resided in dynamically loaded code. For 78 and 7 FNs, points-to analysis for DOM queries were not accurate as the state of the DOM tree and environment variables were necessary for such analysis, respectively. The remaining 350 FNs stemmed from the fact that the JavaScript program used `setTimeout` and `eval` for firing events by generating code at run-time.

5.6.2 HPG Construction

In total, JAW-static generated 498,054,077 nodes and 639,323,601 edges for the 4,836 HPGs, which is 10,756,335 nodes (-2.11%) and 13,338,972 edges (-2.04%) less than JAW (false negatives). Out of the total missing edges, 1,048,172 are ERDDG edges that are critical for modeling events, and the remaining 12,290,800 edges are the AST, CFG, PDG and IPCG edges. Furthermore, JAW-static misses 16,710 edge properties (set on ERDDG registration edges) that mark if an event handler has been triggered at run-time, and that has not been marked with static analysis.

Following additional experiments based on our methodology (§5.1), we logged the fired events that JAW cannot map to their line of code. In total, JAW observed 51,974 events at run-time across 4,836 HPGs, out of which 34,808 were already marked by static analysis and fired dynamically. The remaining 17,166 events trigger at run-time, while not captured by pure static analysis. Out of the 17,166 events, JAW fails to find the corresponding event handlers of 456 events in the code (0.88%), an indication of FN nodes and edges in the HPG. Manual analysis revealed that the reasons for the majority of cases (387 events) is the use of `eval` and `setTimeout` functions with dynamically constructed strings for firing events. The remaining 69 events are not mapped due to the dynamic loading of code and in ways that JAW does not monitor (e.g., loading code from inside iframes).

Finally, we assess the FP and FN edges introduced by the usage of the DOM tree snapshots when performing points-to analysis of DOM queries. In total, JAW encountered 241,428 DOM query selectors in 4,836 HPGs, out of which in 127 selectors (0.05%) JAW imprecisely resolved the DOM element the query is pointing to. To determine the ERDDG dispatch edges, JAW compares the pointers for a total of 87,340 pairs of DOM query selectors against each other (i.e., an event dispatched on one DOM query selector is linked to its event handler that uses another query selector for the event registration). Our evaluation suggests that JAW accurately decides to connect or not to connect a dispatch edge between the dispatch and registration sites in 87,212 cases (decision accuracy of 99.85%), with 56,923 true positives and 30,289 true negatives. In the remaining 128 cases, JAW's decision to create or not to create an edge is inaccurate, with 94 FN and 34 FP edges (decision inaccuracy of 0.15%). Interestingly, we observed that such FP and FN edges may occur for query selectors that are interpreted within 53.7 mili-seconds of page load (on average), and a maximum of 92.5 mili-seconds, which is up to ca. ten times lower than the average access time of all query selectors, i.e., 559.2 mili-seconds. In this experiment, we used run-time program instrumentation to obtain the ground truth for assessing JAW's accuracy in HPG construction. However, such techniques come with performance hits, and are poorly suitable for large HPGs (e.g., in model construction, and vulnerability detection). We believe the impact of JAW's FP and FN edges as a result of DOM snapshots is negligible.

6 Discussion

Properties of Client-side Forgeable Requests. In this paper, we showed that 82% of the web applications have at least one web page with a client-side forgeable request that can be exploited to mount CSRF attacks, suggesting that forgeable requests are prevalent. We also showed that client-side CSRF can be used to mount other attacks, such as XSS and SQLi, which cannot be mounted via the traditional attack vectors. Then, the analysis of forgeable requests suggest that some client-side CSRF templates are more prevalent than others, e.g., in 28.7% of vulnerable applications, the attacker can overwrite a parameter in the request body.

Interesting Properties of Vulnerable Applications. We found that 39 out of 106 targets in our testbed are single page applications (SPA), i.e., 36.7%. We manually examined the 87 vulnerable targets and observed that 44.8% of them are SPA's. Also, we found *exploits* in 17.9% of the tested SPAs

(§5.5). We believe this sheds light into the fact that client-side CSRF instances are more prevalent among SPA applications.

Transfer of Control and Run-time Monitoring. Our evaluation shows that dynamic information increases the transfer of control path by 0.26%. Despite its negligibility, our evaluation shows that dynamic information is fundamental for the identification of the forgeable requests of 14 out of 87 vulnerable applications and three out of seven exploitable applications (an increase of +19.1% and +75%, respectively).

Vulnerability Originates from the Same Code. The manual analysis of the 515 forgeable HTTP requests reveals that each vulnerability originates from different copies of the same code used across various pages. The templates for vulnerabilities range between one to four per application, with a majority of applications (i.e., 78.1%) having only a single template. These facts suggest that developers tend to repeat the same mistake across different pages.

False Positives. We observed that using state values together with traditional static analysis will help to remove spurious execution traces (§5.6). Nevertheless, our extensive manual verification uncovered that 1/516 requests was a false positive due to inaccurate pointer analysis of the `this` statement in dynamically called functions (see §5.3). We observed that such a request is using data values originating from the DOM tree, meaning that 1/83 requests of the `DOM-READ` forgeable request category may be a false positive. We plan to address this shortcoming by incorporating the call-sensitive resolution of the `this` keyword into JAW in the future.

Limitations. The vulnerabilities found in this paper are those captured by our model and traversals. However, it could happen that a forgeable request in the program is not found because the construction of the model is bound by the soundness properties offered by the individual static analysis tools we use for the construction of the property graph, e.g., CFG, PDG, etc. Accurately building these models by static analysis is a challenging task due to the streaming nature of JavaScript programs [43], and JavaScript dynamic code generation capabilities. We point out that, similar to prior work (e.g., see [46]), JAW extracts the code executed by dynamic constructs, i.e., `eval`, `setTimeout` and `new Function()`, as long as the string parameter can be reconstructed statically. As a future work, we plan to replace our extension with a modified JavaScript engine (e.g., VisibleV8 [54]), to provide better support for reflection and such dynamic constructs, and to minimize the potential side effects of function hooking, especially with respect to event handlers. Furthermore, the vulnerabilities discovered in this paper affect those pages that JAW reached with our crawler. However, crawling is a challenging task (see, e.g., [40, 70]) and JAW may have missed pages with vulnerable code. To increase coverage, we plan to provide support for the smooth integration of other crawlers.

Incremental Static Analysis. JAW can reduce by 60% the effort required to analyze client-side JavaScript programs via pre-built symbolic models. When looking at the unique application code, we observe that a large fraction of code is also shared between pages. For example, the 4,836 pages contain in total 104,720 application scripts, of which only 4,559 are unique, suggesting that the shared code of different web pages can be modeled once, and reused through incremental program analysis, a problem we plan to address in the future.

Vulnerability Disclosure. At the time of writing this paper, we are in the process of notifying the affected vendors about our discovery, following the best practices of vulnerability notification (see [85]).

7 Related Work

Request Forgery Vulnerabilities. Request forgery is a widely exploited web vulnerability (see, e.g., [23, 25, 26, 27, 32, 51, 88]) that we can divide into two families: SSRF [68] and CSRF [37, 69]. Research in this area has largely focused on request forgery defenses (e.g., [34, 39, 52, 53, 56, 63, 73, 74]), with very few proposing detection techniques that can help security testing community to uncover CSRF exploits (i.e., [37, 69, 77, 86]). Only a fraction of these works, most notably, Deemon [69], and Mitch [37], went beyond manual inspection by presenting (semi-)automated approaches. As opposed to these works, this paper proposes JAW, a framework to study client-side CSRF vulnerabilities at large-scale based on HPGs and declarative graph traversals.

Property Graphs and Vulnerability Detection. Graph-based analysis of source code has a long history and has a been considered by several researchers (e.g., [33, 41, 57, 71, 91]). Yamaguchi et. al. [91] proposed the notion of CPGs for finding software bugs in C/C++ applications (i.e., a non-web-based execution environment). Backes et. al. [33] later extended this idea to detect vulnerabilities in the server-side of PHP web applications. In contrast to these works, our approach adapts the concept of CPGs to the client-side of web applications, and extends them with dynamic information, i.e., state values (§3.2). Also, existing CPGs are poorly suited for large-scale analyses which is a needed feature to analyze web applications (a web application can have hundreds of pages to analyze, each with thousands of lines of JavaScript code). Backes et. al. [33] needed up to 5 days and 7 hours for a *single* query when analyzing 77M LoC. In comparison, JAW took 3 days (sequential execution) to model and query 228M LoC. This improvement is largely due to the introduction of the new notion of symbolic models for shared third-party code (§5.2). We believe that these contributions are key enablers to use graph-based analyses on web applications, at scale.

Security Analysis of JavaScript Programs. Over the past years, we have seen different techniques for analyzing JavaScript programs (e.g., [38, 42, 44, 46, 61, 62, 67, 82, 83]). To date, these approaches have been mostly applied to XSS [60, 64, 75, 84] and validation flaws [66, 76, 79, 89, 92]). Most notably, Lekies et. al. [60] modified the JavaScript engine in Chromium to enhance it with taint-tracking capabili-

ties, and used a crawler that leverages the modified Chromium to detect DOM-based XSS vulnerabilities. Saxena et. al. proposed Kudzu [75], a tool that performs dynamic taint-tracking to identify sources and sinks in the current execution using a GUI explorer, and then generates XSS exploits by applying symbolic analysis to the detected source-sink data flows. In general, these techniques could be useful to detect client-side CSRF provided their crawler/GUI-explorer can trigger the executions that are connecting sources to sinks. However, crawlers/GUI-explorers often fall short of visiting modern web UIs, providing low code coverage when compared with static analysis techniques. As opposed to approaches like [60, 75], JAW follows a hybrid approach, addressing shortcomings of JavaScript static analysis such as dynamic loading of script tags and point-to analysis for DOM elements.

8 Conclusion

In this paper, we presented JAW, to the best of our knowledge the first framework for the detection and analysis of client-side CSRF vulnerabilities. At the core of JAW is the new concept of HPG, a canonical, static-dynamic model for client-side JavaScript programs. Our evaluation of JAW uncovered 12,701 forgeable client-side requests affecting 87 web applications. For 203 of them, we created a working exploit against seven applications that can be used to compromise the database integrity. We analyzed the forgeable requests and identified 25 different request templates. This work has successfully demonstrated the capabilities of our paradigm for detecting client-side CSRF. In the near future, we intend to use our approach toward additional vulnerability classes.

Acknowledgments

We would like to thank our shepherd Stefano Calzavara and the anonymous reviewers for their valuable feedback.

References

[1] Ast-Flow-Graph library. https://www.npmjs.com/package/ast-flow-graph.

[2] Bitnami application catalog. https://bitnami.com/stacks.

[3] Cypher query language. https://neo4j.com/developer/cypher-query-language/.

[4] Dujs library. https://github.com/chengfulin/dujs.

[5] Escontrol library. https://www.npmjs.com/package/escontrol.

[6] Esgraph CFG generator. https://github.com/Swatinem/esgraph.

[7] Esprima. https://esprima.org/.

[8] Function.prototype.apply(). https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Function/apply.

[9] Function.prototype.call(). https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Function/call.

[10] Headless chromium. https://chromium.googlesource.com/chromium/src/+/lkgr/headless/README.md.

[11] JavaScript language resources. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Language_Resources.

[12] JQuery library. https://jquery.com/.

[13] Library Detector for chrome. https://www.npmjs.com/package/js-library-detector.

[14] Neo4j graph database. https://neo4j.com.

[15] Selenium browser automation. https://www.selenium.dev.

[16] Selenium IDE. https://www.selenium.dev/projects.

[17] Selenium-python. https://selenium-python.readthedocs.io/index.html.

[18] Styx library. https://www.npmjs.com/package/styx.

[19] Usage statistics of content management systems. https://w3techs.com/technologies/overview/content_management.

[20] window.name API. https://developer.mozilla.org/en-US/docs/Web/API/Window/name.

[21] window.open() API. https://developer.mozilla.org/en-US/docs/Web/API/Window/open.

[22] YUI library. https://yuilibrary.com/.

[23] CSRF: Adding optional two factor mobile number in slack, 2016. https://hackerone.com/reports/155774.

[24] Client-side CSRF, 2018. https://www.facebook.com/notes/facebook-bug-bounty/client-side-csrf/2056804174333798/.

[25] Two factor authentication cross site request forgery (CSRF) vulnerability in wordpress. cve-2018-20231., 2018. https://www.privacy-wise.com/two-factor-authentication-cross-site-request-forgery-csrf-vulnerability-cve-2018-20231/.

[26] Account take over in US Dept of Defense, 2019. https://hackerone.com/reports/410099.

[27] Critical CSRF vulnerability on facebook, 2019. https://www.acunetix.com/blog/web-security-zone/critical-csrf-vulnerability-facebook/.

[28] Intent to implement and ship: cookies with SameSite by default, 2019. https://groups.google.com/a/chromium.org/forum/#!msg/blink-dev/AknSSyQTGYs/SSB1rTEkBgAJ.

[29] Intent to implement: Cookie SameSite=lax by default and SameSite=none only if secure, 2019. https://groups.google.com/forum/#!msg/mozilla.dev.platform/nx2uP0CzA9k/BNVPWDHsAQAJ.

[30] SameSite cookie attribute, chromium, blink, 2020. https://www.chromestatus.com/feature/4672634

709082112.
[31] Usage statistics of JavaScript libraries for websites, 2020. https://w3techs.com/technologies/overview/javascript_library.
[32] S. Abdelhafiz. SSRF leaking internal google cloud data through upload function, 2019. https://hackerone.com/reports/549882.
[33] M. Backes, K. Rieck, M. Skoruppa, B. Stock, and F. Yamaguchi. Efficient and Flexible Discovery of PHP Application Vulnerabilities. In *Proceedings of the 2nd IEEE Euro S&P*, 2017.
[34] A. Barth, C. Jackson, and J. C. Mitchell. Robust defenses for cross-site request forgery. In *CCS*, 2008.
[35] A. Barth, J. Weinberger, and D. Song. Cross-Origin JavaScript Capability Leaks: Detection, Exploitation, and Defense. In *USENIX Security*, 2009.
[36] S. Calzavara, M. Bugliesi, S. Crafa, and E. Steffinlongo. Fine-Grained Detection of Privilege Escalation Attacks on Browser Extensions. In *ESOP*, 2015.
[37] S. Calzavara, M. Conti, R. Focardi, A. Rabitti, and G. Tolomei. Mitch: A machine learning approach to the black-box detection of csrf vulnerabilities. In *Proceedings of the IEEE Euro S&P*, 2019.
[38] S. Chandra, C. S. Gordon, J. Jeannin, C. Schlesinger, M. Sridharan, F. Tip, and Y. Choi. Type Inference for Static Compilation of Javascript. In *ACM SIGPLAN Notices*, 2016.
[39] A. Czeskis, A. Moshchuk, T. Kohno, and Helen J. Wang. Lightweight server support for browser-based csrf protection. In *Proceedings of the International Conference on World Wide Web*, 2013.
[40] A. Doupé, L. Cavedon, C. Kruegel, and G. Vigna. Enemy of the State: A State-Aware Black-Box Web Vulnerability Scanner. In *USENIX Security*, 2012.
[41] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. In *ACM Transactions on Programming Languages and Systems*, 1987.
[42] K. Gallaba, A. Mesbah, and I. Beschastnikh. Dońt Call Us, Wéll Call You: Characterizing Callbacks in Javascript. In *Proceedings of the 2015 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, 2015.
[43] S. Guarnieri and B. Livshits. GULFSTREAM: Staged Static Analysis For Streaming JavaScript Applications. In *Proceedings of the USENIX conference on Web application development*, 2010.
[44] B. Hackett, S. Lebresne, B. Burg, and J. Vitek. Fast and Precise Hybrid Type Inference for Javascript. In *PLDI*, 2012.
[45] N. Hardy. The confused deputy: (or why capabilities might have been invented). In *ACM SIGOPS Operating Systems Review*, 1988.
[46] S. H. Jensen, P. A. Jonsson, and A. Møller. Remedying the Eval that Men Do. In *Proceedings of ISSTA*, 2012.
[47] S. H. Jensen, M. Madsen, and A. Møller. Modeling the HTML DOM and Browser API in Static Analysis of Javascript Web Applications. In *Proceedings of the ESEC/FSE*, 2011.
[48] S. H. Jensen, M. Madsen, and A. Møller. Modeling the HTML DOM and browser API in static analysis of JavaScript web applications. In *Proceedings of the ESEC/FSE*, pages 59–69, 2011.
[49] S. H. Jensen, A. Møller, and P. Thiemann. Type Analysis for Javascript. In *Proceedings of the 16th International Symposium on Static Analysis*, 2009.
[50] S. H. Jensen, A. Møller, and P. Thiemann. Interprocedural Analysis with Lazy Propagation. In *International Static Analysis Symposium, Lecture Notes in Computer Science, vol 6337. Springer, Berlin, Heidelberg*, 2010.
[51] M. Johns. The three faces of csrf. talk at the deepsec2007 conference. 2007. https://deepsec.net/archive/2007.deepsec.net/speakers/index.html#martin-johns.
[52] M. Johns and J. Winter. RequestRodeo: Client side protection against session riding, 2006. https://www.owasp.org/images/4/42/RequestRodeo-MartinJohns.pdf.
[53] N. Jovanovic, E. Kirda, and C. Kruegel. Preventing cross site request forgery attacks. In *SecureComm*, 2006.
[54] J. Jueckstock and A. Kapravelos. VisibleV8: In-browser Monitoring of JavaScript in the Wild. In *Proceedings of the ACM IMC*, 2019.
[55] K. Käfer. Cross site request forgery. In *Hasso-Plattner-Institut, Technical report*, 2008.
[56] F. Kerschbaum. Simple cross-site attack prevention. In *SecureComm*, 2007.
[57] D. A. Kinloch and M. Munro. Understanding c programs using the combined c graph representation. In *Proceedings of the International Conference on Software Maintenance*, 1994.
[58] M. S. Lam., R. S. Avaya, and J. D. Ullman. Compilers: Principles, techniques, and tools (2nd edition). In *Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006. ISBN 0321486811*, 2006.
[59] T. Lauinger, A. Chaabane, S. Arshad, W. Robertson, C. Wilson, and E. Kirda. Thou shalt not depend on me: Analysing the use of outdated javascript libraries on the web. *NDSS 2017*, 2017.
[60] S. Lekies, B. Stock, and M. Johns. 25 million flows later: large-scale detection of DOM-based XSS. In *CCS*, 2013.
[61] M. Madsen, B. Livshits, and M. Fanning. Practical Static Analysis of Javascript Applications in the Presence of Frameworks and Libraries. In *Proceedings of the ESEC/FSE*, 2013.
[62] M. Madsen and A. Møller. Sparse Dataflow Analysis with Pointers and Reachability. In *International Static*

Analysis Symposium, Lecture Notes in Computer Science, vol 8723. Springer, Cham, 2014.

[63] Z. Mao, N. Li, and I. Molloy. Defeating cross-site request forgery attacks with browser-enforced authenticity protection. In *13th International Conference on Financial Cryptography and Data Security*, 2009.

[64] W. Melicher, A. Das, M. Sharif, L. Bauer, and L. Jia. Riding out domsday: Towards detecting and preventing dom cross-site scripting. In *NDSS*, 2018.

[65] Mozilla. Introduction to the DOM, 2020. https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model/Introduction.

[66] J. Nicolay, V. Spruyt, and C. D. Roover. Static Detection of User-specified Security Vulnerabilities in Client-side JavaScript. In *PLAS*, 2016.

[67] C. Park and S. Ryu. Scalable and Precise Static Analysis of JavaScript Applications via Loop-Sensitivity (Artifact). In *ECOOP*, 2015.

[68] G. Pellegrino, O. Catakoglu, D. Balzarotti, and C. Rossow. Uses and abuses of server-side requests. In *RAID*, 2016.

[69] G. Pellegrino, M. Johns, S. Koch, M. Backes, and C. Rossow. Deemon: Detecting CSRF with dynamic analysis and property graphs. In *CCS*, 2017.

[70] G. Pellegrino, C. Tschürtz, E. Bodden, and C. Rossow. jäk: Using Dynamic Analysis to Crawl and Test Modern Web Applications. In *RAID*, 2015.

[71] T. Reps. Program analysis via graph reachability. In *Information and Software Technology, 40(11):701–726*, 1998.

[72] G. Richards, S. Lebresne, B. Burg, and J. Vitek. An Analysis of the Dynamic Behavior of Javascript Programs. In *PLDI*, 2010.

[73] P. D. Ryck, L. Desmet, T. Heyman, F. Piessens, and W. Joosen. CsFire: Transparent client-side mitigation of malicious cross-domain requests. In *ESSoS*, 2010.

[74] P. D. Ryck, L. Desmet, W. Joosen, and F. Piessens. Automatic and precise client-side protection against CSRF attacks. In *ESORICS*, 2011.

[75] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song. A symbolic execution framework for JavaScript. In *IEEE S&P*, pages 513–528. IEEE, 2010.

[76] P. Saxena, S. Hanna, P. Poosankam, and D. Song. FLAX: Systematic Discovery of Client-side Validation Vulnerabilities in Rich Web Applications. In *NDSS*, 2010.

[77] H. Shahriar and M. Zulkernine. Client-side detection of cross-site request forgery attacks. In *Proceedings of the IEEE 21st International Symposium on Software Reliability Engineering*, 2010.

[78] S. Sivakorn, I. Polakis, and A. D. Keromytis. The Cracked Cookie Jar: HTTP Cookie Hijacking and the Exposure of Private Information. In *Proceedings of the IEEE Euro S&P*, 2016.

[79] N. Skrupsky, M. Monshizadeh, P. Bisht, T. Hinrichs, V.N. Venkatakrishnan, and L. Zuck. WAVES: Automatic Synthesis of Client-side Validation Code for Web Applications. In *2012 International Conference on Cyber Security*, 2012.

[80] D. F. Somé. EmPoWeb: Empowering Web Applications with Browser Extensions. In *Proceedings of the IEEE S&P*, 2019.

[81] S. Son and V. Shmatikov. The Postman Always Rings Twice: Attacking and Defending postMessage in HTML5 Websites. In *NDSS*, 2013.

[82] T. Sotiropoulos and B. Livshits. Static Analysis for Asynchronous Javascript Programs. In *ECOOP*, 2019.

[83] M. Sridharan, J. Dolby, S. Chandra, M. Schäfer, and F. Tip. Correlation Tracking for Points-To Analysis of Javascript. In *ECOOPs*, 2012.

[84] M. Steffens, C. Rossow, M. Johns, and B. Stock. Don't Trust the Locals: Investigating the Prevalence of Persistent Client-Side Cross-Site Scripting in the Wild. In *NDSS*, 2019.

[85] B. Stock, G. Pellegrino, C. Rossow, M. Johns, and M. Backes. Hey, you have a problem: On the feasibility of large-scale web vulnerability notification. In *USENIX Security*, pages 1015–1032, 2016.

[86] A. Sudhodanan, R. Carbone, L. Compagna, and N. Dolgin. Large-scale analysis & detection of authentication cross-site request forgeries. In *IEEE Euro S&P*, 2017.

[87] A. Sudhodanan, S. Khodayari, and J. Caballero. Cross-Origin State Inference (COSI) Attacks: Leaking Web Site States through XS-Leaks. In *NDSS*, 2020.

[88] R. Walikar. Cross-site port attacks - xspa, 2012. https://ibreak.software/2012/11/cross-site-port-attacks-xspa-part-1/.

[89] M. Weissbacher, W. Robertson, E. Kirda, C. Kruegel, and G. Vigna. ZigZag: Automatically Hardening Web Applications Against Client-side Validation Vulnerabilities. In *USENIX Security*, 2015.

[90] M. West. Incrementally better cookies. 2019. https://tools.ietf.org/html/draft-west-cookie-incrementalism-00.

[91] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck. Modeling and Discovering Vulnerabilities with Code Property Graphs. In *Proceedings of the IEEE S&P*, 2014.

[92] F. Yamaguchi, M. Lottmann, and K. Rieck. Generalized vulnerability extrapolation using abstract syntax trees. In *ACSAC*, 2012.

[93] W. Zeller and E. W. Felten. Cross-site request forgeries: Exploitation and prevention. In *Princeton University*, 2008.

[94] X. Zheng, J. Jiang, J. Liang, H. Duan, S. Chen, and T. Wan. Cookies Lack Integrity: Real-World Implications. In *USENIX Security*, 2015.

A Additional JAW Details

A.1 JAW Semantic Types

Descr.	Type	Example of use
Window URL	WIN.LOC	window.location.hash
Cookie	DOM.COOKIES	doc.cookie
localStorage	LOCAL-STORAGE	doc.localStorage
sessionStorage	SESSION-STORAGE	doc.sessionStorage
postMessage	POST-MESSAGE	addEventListener(evt, h)
Window Name	WIN.NAME	window.name
Document Referrer	DOC.REFERRER	doc.referrer
DOM Attribute	DOM.READ	doc.getElementById('x').value
Client-Side Request	REQ	XMLHttpRequest
Event Dispatch	E-DISPATCH	el.triggerHandler(evt)
Handler Registration	E-REGISTER	el.on(evt, h)
Func. I/O	$o \leftarrow i$	function(i){return $o = g(x)$;}
Func. I/O	$o \sim i$	function(i){if($cond(x)$) return o;}

Table 4: List of semantic types supported by JAW. Types are assigned to constructs representing input sources of a web application, functions that send HTTP requests, dispatch or register events, and functions with inputs/outputs.

Table 4 summarizes the list of semantic types supported by JAW. We can use one semantic type for each of the injection points where the attacker can input data. Semantic types can also be assigned to functions to specify their behavior abstractly, e.g., functions that delegate the dispatch of events or the HTTP requests to low-level browser APIs.

A.2 Library Detection

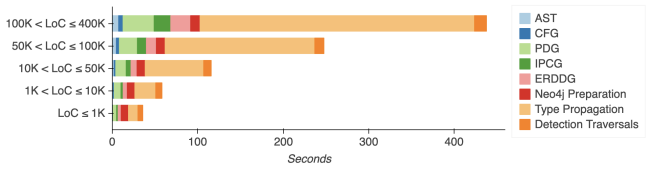
JAW relies on Library Detector [13] to identify the JavaScript libraries used inside a web page. It is used as a bundled script injected by Selenium [15]. Library Detector has a series of pre-defined checks (i.e., usage indicator functions) for each JavaScript library that it supports. It searches for known library signatures inside the execution environment by appling the usage indicator functions. For example, global variables set on the `Window` object by a library are an indicator of the usage of that library. It returns the list of libraries used in the web page. At the time of writing this paper, Library Detector provides support for the detection of 114 different library scripts, including JQuery, React, Angular, and Prototype.

B Additional Evaluation Details

B.1 Testbed (Alphabetically Ordered)

This appendix contains the complete list of the web applications and their versions in our testbed.
AbanteCart 1.2.16, Akeneo 3.2.26, Alfresco Community 201911, Apache Airflow UI 1.10.8, Axelor 5.3.0, Bonita 7.6, CMS Made Simple 2.2.14, CanvasLMS 2020.01.01.05, CiviCRM 5.25.0, Ckan 2.8.0, Collabtive 3.1, Composr 10.0.30, Concrete5 8.5.2, Coppermine 1.6.08, Cotonti 0.9.19, Diaspora 0.7.13.0, Discourse 2.4.5, DokuWiki 20180422c, Dolibarr 11.0.4, DreamFactory 4.2.2, Drupal 8.8.6, ELK 7.6.0, ERPNext 12.9.3, EspoCRM 5.9.1, FatFreeCRM 0.18.1, Fluentd UI 1.10.3, Ghost 3.17.1, Gitlab CE 13.0.3, Grafana 6.5.2, Horde Groupware Webmail 5.2.22, JFrog Artifactory Open Source 6.19.1, JasperReports 7.5.0, Jenkins 2.204.1, JetBrains YouTrack 2019.3.62973, Joomla 3.9.18, Kibana 7.5.1,

Figure 6: Average time required for JAW to construct and analyze a hybrid property graph categorized by lines of code (LoC).



Kong Admin UI 0.4.1, Kubeapps 1.9.0, Let's Chat 0.4.8, Liferay 7.2.1, LimeSurvery 4.2.5, Live Helper Chat 3.27, LotusCMS 3.0.5, Magento 2.3.5, Mahara 19.10.1, Mantis 2.24.1, Matomo 3.13.1, Mattermost 5.14.0, Mautic 2.16.2, MediaWiki 1.34.1, Moalyss 7.3.0.0, Modx 2.7.3pl, Moodle 3.8.3, MyBB Forum 1.8.22, Neos 5.2.0, OXID eShop 6.2.1, Odoo 13.0.20200515, Open Atrium 2.646, Open edX ironwood.2.8, OpenCart 3.0.3.2, OpenProject 10.5.1, Openfire 4.4.4.1, OrangeHRM 4.4, OroCRM 4.1.4, Osclass 3.9.0, Parse Server 4.2.0, ParseDashboard 2.0.5, Phabricator 2020.21, Pimcore 6.6.4, Plone 5.2.1, Pootle 2.8.2, PrestaShop 1.7.6.2, ProcessMaker Community 3.3.6, ProcessWire 3.0.148, Prometheus 2.18.1, Publify 9.1.0, Re:dash 8.0.0, Redmine 4.1.1, Report Server Community 3.1.1.6020, Report Server Enterprise 3.1.1.6020, ResourceSpace 9.2.14719, ReviewBoard 3.0.17, Roundcube 1.4.5, SEO Panel 4.3.0, Shopware 6.1.0, Silverstripe 4.5.2, Simple Machines Forum 2.0.17, SonarQube 8.2.0.32929, Spree 4.1.6, SugarCRM 6.5.13, SuiteCRM 7.1.1, TestLink 1.9.20, Tiki Wiki CMS Groupware 21, Tiny Tiny RSS 202006, Trac 1.5.1, Typo3 10.4.3, Weblate 4.0.3, Webmail Prop PHP 8.3.20, Wordpress 5.4.1, Xoops 2.5.10, Zurmo 3.2.7, eXo Platform 5.3.0, ownCloud 10.4.1, phpBB 3.3.0, phpList 3.5.4, and phpMyAdmin 5.0.1.

B.2 Run-time Performance of JAW

We deployed the web applications under evaluation on a desktop computer (running MacOS Mojave 10.14.3 on an Intel Core i5 with 2.4 GHz, 16 GB RAM, and a SSD), and performed the data collection step (§4.1). We let JAW run for a maximum of 24 hours on each web application, although after a few hours the data collection module typically does not find any new URLs. Then, we imported the collected data on our own server (running Ubuntu 18.04 on an Intel(R) Xeon(R) CPU E5-2695 v4 with 2.10 GHz and 72 cores, 252 GB RAM), and instantiated JAW with the data to find client-side CSRF vulnerabilities. We log all processing times for throughput evaluation. Figure 6 depicts the average processing time for each tool component in order to construct and analyze a HPG. As shown in the figure, the processing time increases as the LoC grows. The least time consuming operations are AST and intra-procedural CFG construction. JAW also a incurs a preparation delay in order to import the constructed property graph into a Neo4j database which typically lasts around 8-11 seconds based on the LoC. The most time consuming operation is the semantic type propagation.