TRANSPARENT: Taint-style Vulnerability Detection in Generic Single Page Applications through Automated Framework Abstraction

Senapati Diwangkara
Johns Hopkins University
diwangs@cs.jhu.edu

Yinzhi Cao
Johns Hopkins University
yinzhi.cao@jhu.edu

***Abstract*—Single Page Application (SPA) frameworks allow developers to build complex web applications in a single HTML page with high-level components (e.g., search box). One research problem for SPAs is how to detect taint-style vulnerabilities, because the SPA framework reintroduces insecure DOM APIs in a new format, such as SPA component parameters as taint sinks. Although previous work has focused on improving vulnerability detection in SPAs, to the best of our knowledge, they rely heavily on hard-coded taint sinks, which not only need to be manually curated for each different SPA framework but may also miss certain insecure SPA APIs, introducing false negatives in detected vulnerabilities.**

In this paper, we present TRANSPARENT, an SPA vulnerability detection tool that automatically abstracts SPA frameworks using a combination of static and dynamic analysis to reveal framework-specific sinks, thus facilitating end-to-end static vulnerability detection. TRANSPARENT first performs a backward taint analysis from a list of insecure DOM APIs up to the framework interface to reveal which part of the interface could taint the DOM API. This automated framework abstraction is done once per SPA framework. Then, TRANSPARENT finds dataflow paths between the detected SPA sinks and attacker-controlled sources to detect taint-style vulnerabilities in each application. We evaluated TRANSPARENT against a database of GitHub repositories and found 11 zero-day vulnerabilities, including a repository with 24k+ GitHub stargazers and 30 million requests/month. So far, four zero-day vulnerabilities has been fixed and/or acknowledged by their developers. During our evaluation, TRANSPARENT found a total of 19 intermediate SPA sinks from the three most widely used SPA frameworks, Vue, React, and Angular. 14 of the newly discovered sinks are not listed by the CodeQL standard library, the state-of-the-art static analysis tool.

I. INTRODUCTION

Frontend frameworks like Angular [1], React [37], and Vue [42] help web developers build complex web pages by raising the level of abstraction of their building blocks, from individual HTML tags to a logical component, such as a search box. This is done through an abstraction runtime that runs on the browser's JavaScript engine, which instruments the browser's DOM API while, in turn, providing a framework API to be used by the developer. Using such frameworks, the complexity of a website could increase to the point that it could be developed to run as a standalone application outside of a traditional browser environment, requiring less (sometimes even zero) contact with a server [11], [3]. Such a paradigm is known as Single Page Application (SPA) [2], named after the fact that the application only has a single HTML page, leaving all the page-handling logic contained in JavaScript.

Despite its differences, SPA is still prone to the same web taint-style vulnerabilities as its classic counterparts. There are two classic, complementary methods for vulnerability detections: dynamic and static. The former [22], [41], [31] is more accurate with fewer or sometimes zero false positives but needs deployment, e.g., environment setup, and input exercise to increase code coverage. The latter is the focus of the paper, which has more false positives, but is scalable without runtime support. The inclusion of an SPA framework makes static taint-style vulnerability detection more difficult, since malicious input could flow through said runtime to an insecure DOM API.

One natural way to tackle the obstacle is with whole-program analysis: analyzing the SPA runtime together with the application of interest. However, whole-program analysis is generally not a scalable approach, and doing it on JavaScript is tricky due to its dynamic nature [30], [20]. Additionally, to the best of our knowledge, none of them are equipped to comprehensively handle the syntaxes present in an SPA framework, like HTML templating language. Another way adopted by recent static vulnerability detection, such as CodeQL [9] and ReactAppScan [21], is to focus their analysis solely on the SPA with an abstract of the framework, e.g., using framework-specific sinks (called SPA sinks or stubs). However, they rely on hard-coding the SPA sinks per framework and with manual summarization. This is not a comprehensive approach because it is challenging to enumerate sinks for different SPA frameworks, especially with manual efforts.

In this paper, we design and implement a system called TRANSPARENT,[1] to comprehensively detect taint-style vulnerabilities on SPAs built on top of many different SPA frame-

[1]TranSPArent: Trace-assisted Analysis for SPA Impairment

Network and Distributed System Security (NDSS) Symposium 2026
23-27 February 2026, San Diego, CA, USA
ISBN 979-8-9919276-8-0
https://dx.doi.org/10.14722/ndss.2026.241721
www.ndss-symposium.org

works. Central to our approach is the key idea of *automated framework abstraction*: TRANSPARENT first models the SPA runtime to uncover framework-specific intermediate SPA sinks that could potentially be used by an SPA built on top of it. This analysis is done once per framework, and the resulting SPA sinks are then used with a commercial-off-the-shelf SPA static vulnerability detection tool to mine vulnerabilities within them.

While the idea is intuitively simple, the automated framework abstraction faces two major challenges:

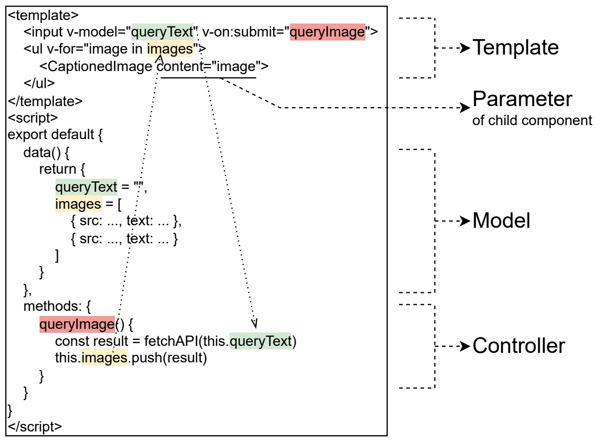
- *Taint-rule diversity*, which is caused by the fact that SPA frameworks contain many dynamic language features. We introduce an automated technique within TRANSPARENT to deal with the SPA runtime's reliance on dynamic features called *autostitch*. The key idea is to use stack traces produced by the SPA runtime's test suite to 'stitch' an incomplete call graph.
- *Sink diversity*, which is the fact that we have multiple SPA frameworks, each with numerous different methods (e.g., JavaScript and HTML syntaxes) of exposing the raw DOM API to developers. TRANSPARENT performs backward dataflow analysis of SPA frameworks from traditional insecure DOM APIs to JavaScript interfaces for SPA sinks with JavaScript syntax. Then, TRANSPARENT analyzes the input and output of the HTML template language transpiler to detect SPA sinks with HTML syntax based on their JavaScript-syntax SPA sink counterparts.

We validated our idea by applying TRANSPARENT to a dataset of crawled GitHub repositories that use the top three most popular SPA frameworks to date (Angular, React, and Vue) and found 11 zero-day vulnerabilities. In the course of doing so, TRANSPARENT discovered 19 intermediate SPA sinks, 14 of which are missed by the hard-coded SPA sinks used by the state-of-the-art Single-Page Application static-analysis tool, CodeQL. Our contributions are twofold:

- TRANSPARENT - We designed and implemented a Single Page Application vulnerability detection tool that could handle both sink and taint-rule diversity to be applied to many popular SPA frameworks. Our implementation is published with an open-source license.
- Zero-day Vulnerabilities - We found 11 zero-day vulnerabilities by applying TRANSPARENT against a dataset of GitHub repositories of SPAs based on Angular, React, and Vue. We responsibly disclosed our findings to the maintainer according to their security policy.

II. BACKGROUND

SPA frameworks vary in implementation, but they are functionally similar. To do a generic analysis, this section will present a structure of how a Single Page Application is generally written. SPA framework's inner workings revolve around the concept of a high-level *component*. This component is defined once and could be instantiated many times. One could think of this role as a special case of object-oriented programming[2] (OOP) where the component is a class that could be instantiated into objects that are the concrete UI elements, which are called *views*. The SPA runtime provides a special function called a *render function* to instantiate a component into view. While *component* and *view* technically refer to distinctly different items, this paper will commit a slight abuse of definition by using the word *component* to refer to both. A component mainly consists of *templates*, *parameters*, *models*, and *controllers*. Figure 1 depicts all these different parts in an abridged example of a Vue page.

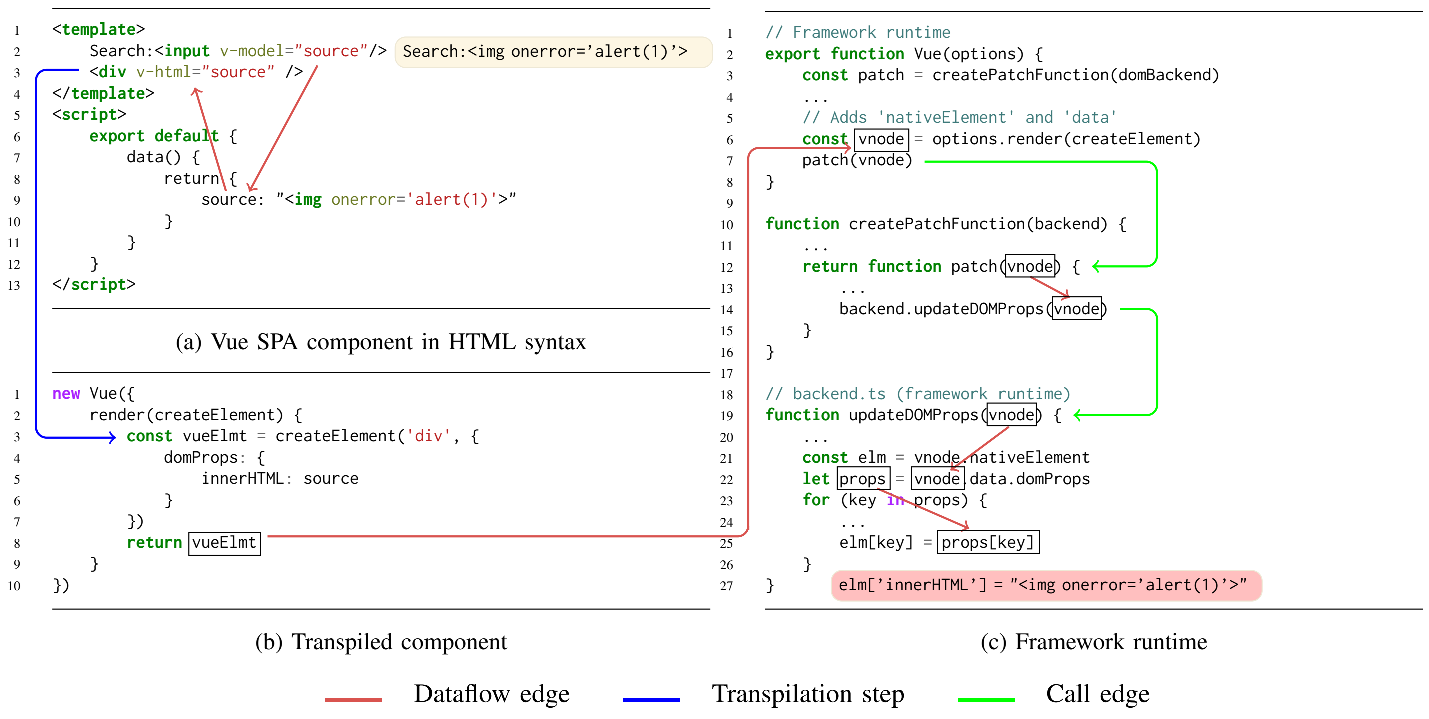


```
<template>
   <input v-model="queryText" v-on:submit="queryImage">
   <ul v-for="image in images">
      <CaptionedImage content="image">
   </ul>
</template>
<script>
export default {
   data() {
      return {
         queryText = "",
         images = [
            { src: ..., text: ... },
            { src: ..., text: ... }
         ]
      }
   },
   methods: {
      queryImage() {
         const result = fetchAPI(this.queryText)
         this.images.push(result)
      }
   }
}
</script>
```

Fig. 1: Illustrative example of an image searching page in Vue that breaks down all the different parts of a component: template, parameter, model, and controller. Dotted arrows show how models are kept in sync with the template after initialization.

- *Template* is the declarative structure of the component, composed of either a primitive component (which mostly mirrors a native HTML element like `div`), component interpolation, or yet another user-defined component. A page is also a component, serving as the root of a component tree (which mimics a DOM tree).
- *Parameter* is the input data to fill the template with, just like HTML element attributes or properties of a native HTML file. We will call it the component's *parameter* to signify its difference from HTML concepts. This concept resembles a constructor parameter in OOP.
- *Model* defines an internal state of components. This resembles a property in OOP. A model is usually bound to some parameter within the template and kept in sync by the SPA framework: a change in the model would change the template parameter and *vice versa*.
- *Controller* is the set of imperative actions that the component could do (e.g., event-handling), which resembles a method in OOP. A component also has a special constructor method that would be called internally by the render function on instantiation.

[2]This comparison is purely for explanatory purposes; one could have an SPA runtime whose idiom revolves around other programming paradigms, like React's functional components.



```
1  <template>
2      Search:<input v-model="source"/>    Search:<img onerror='alert(1)'>
3      <div v-html="source" />
4  </template>
5  <script>
6      export default {
7          data() {
8              return {
9                  source: "<img onerror='alert(1)'>"
10             }
11         }
12     }
13 </script>
```

(a) Vue SPA component in HTML syntax

```
1  new Vue({
2      render(createElement) {
3          const vueElmt = createElement('div', {
4              domProps: {
5                  innerHTML: source
6              }
7          })
8          return vueElmt
9      }
10 })
```

(b) Transpiled component

```
1  // Framework runtime
2  export function Vue(options) {
3      const patch = createPatchFunction(domBackend)
4      ...
5      // Adds 'nativeElement' and 'data'
6      const vnode = options.render(createElement)
7      patch(vnode)
8  }
9
10 function createPatchFunction(backend) {
11     ...
12     return function patch(vnode) {
13         ...
14         backend.updateDOMProps(vnode)
15     }
16 }
17
18 // backend.ts (framework runtime)
19 function updateDOMProps(vnode) {
20     ...
21     const elm = vnode.nativeElement
22     let props = vnode.data.domProps
23     for (key in props) {
24         ...
25         elm[key] = props[key]
26     }
27 }      elm['innerHTML'] = "<img onerror='alert(1)'>"
```

(c) Framework runtime

Fig. 2: A simplified example of a zero-day vulnerability found by TRANSPARENT, which contains the (a) developer-written code, (b) transpiled code, and (c) framework code. Specifically, the figure shows that a dataflow from the source (Line 9 of a) to an SPA sink (`v-html` at Line 3 of a) and then to a DOM sink (Line 25 of c) through the transpiled code (b). The yellow-shaded box signifies the user-controllable source: a text input, and the red-shaded box signifies the DOM sink, representing one execution of line 25.

III. Overview

This section will give a brief overview of the problem and challenges that TRANSPARENT is solving. This includes our threat model and a real-world code snippet to motivate the problem.

A. Threat Model

We consider a threat model similar to classic taint-style client-side web vulnerabilities, where a payload flows from taint sources down to taint sinks to produce a vulnerability. We assume that an adversary has control over a user input on the page (taint source), such as a URL or a form, whose content could flow down to an insecure client API (taint sink). In contrast to classic client-side web vulnerabilities, however, we only consider client APIs that come from an SPA framework and are a necessary condition for an exploit, instead of the browser runtime. Throughout this paper, we will use the phrase "SPA sink" to refer to the former and "DOM sink" to refer to the latter.

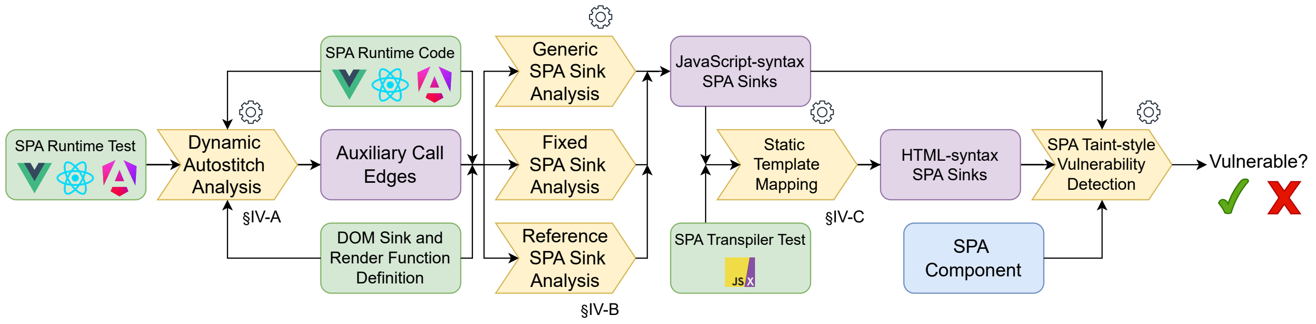
Such an SPA sink could be used alone (e.g., a function to modify the DOM directly) or in conjunction with other functions (e.g., a function that returns a raw DOM element from a component). This also implies the absence of sanitization within the framework runtime itself. We consider all taint-style client-side web vulnerabilities, like XSS and open redirects, to be in scope. Vulnerabilities that do not rely on client weakness (e.g., server-side rendering vulnerabilities) are considered out of scope.

In light of prior works [21], we also consider two kinds of vulnerabilities: application-level and package-level. Application-level vulnerabilities are end-to-end vulnerabilities whose taint sources are end-user accessible (e.g., URL, form). Package-level vulnerabilities, on the other hand, are vulnerabilities whose sources are component parameters. Such vulnerabilities come from a library that does not implement sanitization internally.

B. A Motivating Example

We motivate the necessity of our solution by illustrating a simplified version of a zero-day vulnerability found by TRANSPARENT in Figure 2, which shows developer-written code (Figure 2a), transpiled code (Figure 2b), and framework-provided code (Figure 2c). The vulnerability is located in BiliBili-Evolved [18], a popular (24k+ GitHub stargazers, 30 million requests/month [24]) browser script to enhance the functionality of the BiliBili video streaming website. We responsibly reported the vulnerability, which was first acknowledged and then fixed by the code developers. In this code snippet, a vulnerable user input flows from an `input` component at line 2 in Figure 2a to an SPA sink at line 3 in Figure 2a and then an insecure DOM API at line 25 in Figure 2c.

Analyzing the full taint flow of this vulnerability is challenging due to the scale of the SPA framework and its taint-rule diversity. The challenge of taint-rule diversity is manifested in two parts. First, this vulnerability relies on dynamic language



SPA Runtime Test
Dynamic Autostitch Analysis
§IV-A
SPA Runtime Code
Auxiliary Call Edges
DOM Sink and Render Function Definition
Generic SPA Sink Analysis
Fixed SPA Sink Analysis
Reference SPA Sink Analysis
§IV-B
JavaScript-syntax SPA Sinks
Static Template Mapping
SPA Transpiler Test
§IV-C
HTML-syntax SPA Sinks
SPA Taint-style Vulnerability Detection
SPA Component
Vulnerable?

Fig. 3: TRANSPARENT system design. Green boxes indicate framework abstraction input (defined once per framework). Blue boxes indicate vulnerability detection input (defined once per application). Purple boxes indicate intermediate output.

features. An example of this is the factory function at line 10 in Figure 2c that conceals the call edge between line 7 and line 12. Second, the vulnerability relies on two syntaxes: the Vue template syntax in Figure 2a and JavaScript in the framework runtime in Figure 2c. To the best of our knowledge, no commercial-off-the-shelf static analysis tool could handle these challenges and is thus unable to find this vulnerability.

In lieu of detecting the full taint flow, commercial-off-the-shelf static analysis tools narrow the scope of detecting vulnerabilities by only analyzing the component. To do this, they hardcode the SPA sinks, thus giving rise to the sink diversity problem. In this example, there are two SPA sinks: line 3 in Figure 2a (`v-html` parameter) and line 5 in Figure 2b (`domProps.innerHTML`), which represent two different styles to write raw HTML content in Vue. As far as we know, no commercial-off-the-shelf static analysis tool could detect the latter SPA sink.

To iron out both of these problems simultaneously, TRANSPARENT performs a staged analysis by first modeling the SPA framework in order to automatically produce an abstraction of each framework in the form of intermediate SPA sinks. This approach is more scalable since the framework analysis is only done once per framework, and more flexible since its resulting SPA sinks could then be used by a commercial-off-the-shelf static analysis tool that traditionally uses hard-coded SPA sinks for in-component taint analysis.

To tackle the first problem of dynamic language features that is present at line 10 of Figure 2c, in particular, TRANSPARENT runs the unit tests that come with the Vue runtime to complete the call edge that is missed by static analysis tools. The approach is called "autostitch" in the paper. After that, TRANSPARENT does a backward-taint analysis with the auxiliary call edges from a DOM sink back to an SPA interface to find the list of SPA sinks that the developer could import.

Then, to tackle the second problem of multiple syntaxes, we leverage the insight that HTML-syntax templates have a JavaScript-syntax equivalent after transpilation. In the context of Figure 2, Figure 2a will be transpiled into Figure 2b. To automatically look for this relationship, TRANSPARENT employs a pattern matching analysis that is applied to the Vue transpiler unit tests to figure out how HTML-syntax SPA sink gets transpiled to its corresponding JS-syntax SPA sink.

IV. DESIGN

In this section, we describe the core design of TRANSPARENT and how it adopts automated framework abstraction as a part of taint-style vulnerability detection on SPA. Figure 3 illustrates the overall system design. In short, TRANSPARENT addresses the taint-rule diversity (both regarding dynamic language features and multiple syntaxes) and sink diversity problems in the following three successive analyses:

(A) *Dynamic Autostitch Analysis*: SPA runtime contains dynamic language features, such as higher-order functions, which are absent from the call graph produced by a commercial-off-the-shelf static analysis tool.
To solve this issue, TRANSPARENT employs a dynamic analysis subsystem called autostitch. The subsystem's basic premise is to use stack traces to complete a deficient call graph. This subsystem produces auxiliary call edges to reveal taint paths that are not discovered by ordinary commercial-off-the-shelf static analysis tools.

(B) *Static Taint Path Analysis*: SPA runtime contains a multitude of SPA sinks that are traditionally listed manually by commercial-off-the-shelf tools, leading to undertainting.
To solve this sink diversity issue, TRANSPARENT employs an automated static analysis subsystem that analyzes the SPA runtime for a taint path from a DOM sink back to the render function. It then deduces which specific part of the render function parameter the SPA sink is located in by analyzing the parameter object shape, while also taking into account the parameter key translation logic that may exist in a given SPA runtime. This subsystem will produce a list of JavaScript-syntax SPA sinks.

(C) *Static Template Mapping Analysis*: SPA runtime is built using JavaScript, but the SPA component is idiomatically written in an HTML-like template language. Due to this multiple syntax problem, commercial-off-the-shelf static analysis tools are unable to detect the full taint path from the template language to a DOM sink.
To solve this issue, TRANSPARENT exploits the fact that SPA components written in HTML-like template language always have a corresponding JavaScript-syntax equivalent by means of transpilation. TRANSPARENT

then employs a pattern-matching technique to figure out the mapping between the HTML-syntax attribute and the previously discovered JavaScript-syntax parameter to produce a list of HTML-syntax SPA sinks.

TranSPArent then uses the resulting JavaScript-syntax and HTML-syntax SPA sinks to detect vulnerabilities in SPA components by utilizing them as additional taint rules in a commercial-off-the-shelf static analysis tool. We detail each of these mechanisms in the following subsections below.

A. Dynamic Autostitch Analysis

The first analysis stage of TranSPArent is the dynamic autostitch analysis. In order for TranSPArent to handle dynamic function patterns that manifest as gaps in the call graph, TranSPArent automatically 'stitches' this gap using data points obtained from the SPA runtime. There are three dynamic function patterns that TranSPArent addresses that are particularly prevalent in SPA runtime:

1) Higher-order Function: By far the most prevalent pattern in SPA runtime that causes the problem of incomplete call graph is higher-order functions. To achieve goals such as platform modularization and scheduling, SPA runtime often relies on higher-order function patterns such as a factory function. Unfortunately, commercial-off-the-shelf static-analysis tools lack the capability of generating call edges from such scenarios. Compounding the problem, each SPA runtime employs a different internal architecture, making them challenging to solve all at once.

Fortunately, the codebase of an SPA runtime, like that of large and active JavaScript projects, always comes with an accompanying test suite. We observe that this test suite always tests the end-to-end rendering pipeline at least once. Intuitively, its execution will produce at least one stack trace with the complete call edges from a render function to a DOM sink and could be used to augment an incomplete call graph.

We detail our method in Algorithm 1. The autostitch algorithm takes the framework source code (`C`) and its test suite (`T`) as input (line 1). It will return the set of missing call edges (`S`), which is initialized as empty (lines 2 and 3). We start by searching and marking DOM sinks in a given SPA runtime codebase[3] by inserting a `console.trace` statement antecedently as a cue to print the stacktrace when said DOM sinks are executed. After that, TranSPArent runs `T`, parses, and collects `stacktraces` from standard error (line 4). For each unique `stacktrace` in the `stacktraces` set, TranSPArent examines the function invocation within it. For each successive invocation (i.e., `stacktrace[i]` and `stacktrace[i+1]`), TranSPArent will check whether the former has a call edge to the latter's enclosing function (line 7-9). TranSPArent uses the filename and line number locator to search for the correct caller and callee function. If they are not reachable, TranSPArent will append them to `S` (line 10) and return after all `stacktraces` have been processed (line 14). From the set of missing call edges `S`, TranSPArent

Algorithm 1: Autostitch algorithm pseudocode

```
Input: $C$ = Framework source code
$T$ = Framework test suite
Output: $S$ = Set of (missing) call edges
1  $S \leftarrow \{\}$
2  $stacktraces \leftarrow \texttt{test}(C, T)$
3  foreach $stacktrace$ in $stacktraces$ do
4      for $i \leftarrow 0$ to $stacktrace.length - 1$ do
5          $fCaller \leftarrow stacktrace[i]$;
6          $fCallee \leftarrow enclosingFunction(stacktrace[i+1])$;
7          if not $\texttt{reachable}(fCaller, fCallee)$ then
8              $S.\texttt{append}(fCaller, fCallee)$;
9          end
10     end
11 end
12 return $S$
```

will add the call edges to the analysis and also add a dataflow edge between the argument of the caller and the parameter of the callee for later analysis stages.

2) Component-generated Function: Another challenge that needs to be tackled is the component-generated function. Some SPA runtimes require a dynamically generated component-specific function to accomplish their rendering. As such, this kind of function cannot be analyzed statically unless the relevant component is known beforehand. Since TranSPArent's automated framework abstraction is meant to be a component-agnostic approach, a workaround is needed to tackle this problem.

TranSPArent solved this issue by employing a heuristic where, if a function in the stack trace cannot be found in the codebase (suggesting that it is dynamically generated), TranSPArent skips said function and stitches the function call to the next function below it, ensuring that the dataflow path remains intact.

3) Bound Function: Yet another challenge that needs to be tackled is regarding bound functions. JavaScript has a built-in method called `bind()` that attaches to a function object. It accepts arguments typically passed to the base function to dynamically create an altered version of said function with a parameter value fixed to the previously passed argument, which is referred to as a 'bound function'. While the stack trace provided by the higher-order function analysis shows the correct call edge relationship between the caller and callee, creating a dataflow edge between the caller's argument and the callee's parameter for a bound function statically or dynamically from the stack trace is tricky because the location of the invocation is different from where the argument is defined. To address this, TranSPArent employs a heuristic where we treat function binding as a function invocation. We note that the idiomatic usage of a bound function within an SPA runtime is for scheduling; that is, a bound function is created to be inserted into a scheduling queue for later execution. We observe that this function and its parameter do not get read or modified after binding; thus, this heuristic is guaranteed not to cause any side effects. Unlike higher-order and component-generated functions, this heuristic is

[3]TranSPArent excludes auxiliary components like DevTool

TABLE I: A List of sensitive DOM APIs. `nativeAttr` and `nativeProps` correspond to any attributes and properties, respectively, of the underlying HTML tag. For the list of sensitive attribute and properties, refer to Table II.

Sensitive DOM API	Kind
`window.eval()`	Static DOM sink
`window.open()`	Static DOM sink
`document.write()`	Static DOM sink
`document.writeln()`	Static DOM sink
`elm.appendChild()`	Static DOM sink
`elm.insertBefore()`	Static DOM sink
`elm.appendChild()`	Static DOM sink
`elm.insertAdjacentHtml()`	Static DOM sink
`elm.<nativeProp>`	Static DOM sink
`elm[<nativeProp>]`	Dynamic DOM sink
`Object.defineProperty(elm, <nativeProp>)`	Dynamic DOM sink
`elm.setAttribute()`	Dynamic DOM sink
`elm.setAttributeNS()`	Dynamic DOM sink

framework-independent and automatically applied to the taint-tracking process as auxiliary taint rules.

Thus, with all the auxiliary taint rules ready, TRANSPARENT could proceed to the next stage of the analysis.

B. Static Taint Path Analysis

The second analysis stage of TRANSPARENT is the static taint path analysis. The goal of this stage is to discover JavaScript-syntax SPA sinks in a given runtime by analyzing the backward taint path from the DOM sinks within it, which is defined as a DOM API that interprets its input as HTML or URL. The SPA sink is then manifested as a component parameter in the render function in the form of a key-value pair.

The key challenge is that SPA sinks are still very diverse even when only considering the JavaScript-syntax one, as there are many different ways a DOM sink can be triggered by a given SPA sink. To principally address this problem, we identify common patterns of how a JavaScript-syntax SPA sinks relate to its underlying DOM sinks. To this end, we devise a taxonomy of two types of DOM sinks and three types of SPA sinks to facilitate TRANSPARENT's analyses in uncovering the different ways SPA sinks might form.

First, we divide DOM sinks into two categories depending on which syntax is used to refer to the DOM property. The first is a ① *static DOM sink*, which either does not refer to any DOM property explicitly or uses property accessor syntax whose key could always be determined statically. Examples include `elm.insertAdjacentHtml()` and assignment to dot-notation property, such as `elm.innerHTML`. The second is a ② *dynamic DOM sink*, which uses property accessor syntax whose key might not be determined statically. Examples include assignment to bracket-notation property `elm[<nativeProp>]` or a method call `elm.setAttribute(<nativeAttr>, val)`. The complete list is in Table I.

Second, we divide SPA sinks into three categories depending on two axes of criteria: how the underlying element is accessed and how the parameter key flows. In particular, since

TABLE II: List of sensitive attributes and properties. Sensitive attributes are used to compare with HTML-syntax component and sensitive properties are used to search for DOM sinks.

Sensitive attrs and props	Type	Interpret string as
`src`	Attrs & Props	URL
`href`	Attrs & Props	URL
`action`	Attrs & Props	URL
`formaction`	Attrs & Props	URL
`data` (object)	Attrs & Props	URL
`code` (embed)	Attrs & Props	URL
`srcdoc` (iframe)	Attrs & Props	URL
`xlink:href`	Attrs & Props	URL
`location` (window)	Props	URL
`innerHTML`	Props	HTML
`outerHTML`	Props	HTML

a basic SPA component encapsulates exactly one DOM element, this underlying DOM element could either be accessed directly or indirectly (i.e., through an interface). Similarly, for the second criterion, when a component parameter is set to a certain value, while its payload (i.e., its value) always flows directly to a DOM sink, the name of this parameter (i.e., its key) could either flow directly or indirectly to said DOM sink. Note that if the underlying DOM element could be accessed directly, the parameter will always flow directly to it. Thus, the resulting three categories are as follows:

- *Generic SPA sink* - This type of SPA sink does not have direct access to the underlying DOM element and has a parameter setting mechanism whose key flows directly to the DOM sink. This SPA sink implies the usage of a dynamic DOM sink underneath.
- *Fixed SPA sink* - This type of SPA sink also does not have direct access to the underlying DOM element and has a parameter setting mechanism whose key does not flow directly to the DOM sink. This SPA sink implies the usage of a dynamic DOM sink with a literal string or a static DOM sink underneath.
- *Reference SPA sink* - This type of SPA sink provides direct access to the underlying DOM element (i.e., providing a reference) that could be abused by its user.

Accordingly, TRANSPARENT is equipped with three different analyses for each category of the SPA sinks:

1) Generic SPA Sink Analysis: A generic SPA sink has a parameter key that flows directly into a DOM sink, which implies the usage of a dynamic DOM sink. For example, given a render function `render('a',{href:"blank"})` which creates a blank link in the page, it will need to use a dynamic DOM sink, for example, `elm['href']='blank'`. The running example from Figure 2 is one instance of a generic sink.

To detect a generic SPA sink, TRANSPARENT does two separate sub-analyses: parameter value taint analysis and parameter key dataflow analysis. The general idea is that since parameter key flows directly into the DOM sink, TRANSPARENT needs to find this dataflow path and figure out if there is any string modification along the way.

a) Generic Parameter Value Analysis: Before analysis of the parameter key could occur, TRANSPARENT needs to know whether a given DOM sink could be tainted by the render function in the first place. To do this, TRANSPARENT utilizes a commercial-off-the-shelf static analysis tool to find a path from a dynamic DOM sink back to the render function.

A render function usually has parameters that specify what DOM property to modify and with what value. This parameter could take many forms, but one common shape is a key-value JavaScript object. In addition to figuring out whether a taint path exists, TRANSPARENT also checks the property read syntax along its path to figure out which part of the object specifically that will flow into the DOM sink.

In our example, both of our HTML element property value (image with `onerror`) and its key (`innerHTML`) flow from a nested object: they are both located in the `domProps` property of the `vnode.data` object. The payload comes from the property value, and the property name comes from the key. This key-value format is generally true for all generic SPA sinks.

b) Generic Parameter Key Analysis: Aside from the path of the parameter value, TRANSPARENT also pays attention to the parameter key in order to accurately locate the source of the payload within the render function.

One of the reasons TRANSPARENT needs a separate analysis for parameter key is to analyze them for translation logic. The parameter name that SPA runtime gives is sometimes different than the property name of the DOM element. While this logic does not exist for Vue (thus not present in the running example of Figure 2), in React, for example, the `formaction` attribute for HTML forms is translated to `formAction` with a capital A in the framework. While only having one character difference, this is enough to cause a false negative in a static analysis engine.

In our observation across different SPA frameworks, we observe two dominant patterns of translation logic inside the framework runtime.

The first translation logic is a JavaScript runtime built-in string modification function, like `replace()`. To detect this, TRANSPARENT analyzes the parameter name path for string modification functions invocation and collects them in a list.

The second translation logic is dynamic translation, which uses a "dictionary" object. In this translation logic, the runtime keeps a dictionary mapping between the expected SPA parameter key and the corresponding native DOM property key translation. This dictionary is always accessed dynamically since the resulting value could only be obtained at runtime after we have a concrete parameter name to translate. Because of its dynamic nature, it is usually accessed through a dynamic syntax, such as bracket notation (e.g., `dict[parameterName]`. TRANSPARENT detects this in the parameter key path and collects them.

After TRANSPARENT has listed the sequence of non-value preserving operations, be it string modification or dynamic translation, TRANSPARENT then deduces the framework parameter name by applying the reverse operation to a fixed list of insecure DOM properties. For dynamic translation, for example, TRANSPARENT constructs a reverse mapping and deduces the sensitive parameter key based on this reverse mapping. Thus, after applying the reverse operation, TRANSPARENT can list the insecure framework parameter key.

2) Fixed SPA Sink Analysis: To detect a fixed SPA sink, TRANSPARENT also does the same two separate sub-analyses: parameter value taint analysis and parameter key dataflow analysis. The analysis of Fixed SPA Sink is different than its generic counterparts since the parameter key does not flow directly into the DOM sinks. Thus, TRANSPARENT does a different approach in these two sub-analyses:

a) Fixed Parameter Value Analysis: TRANSPARENT analyzes the parameter value of a fixed SPA sink in a similar fashion to a generic SPA sink, with one key addition: analysis of guard node condition. The intuition is that since the parameter key does not flow directly into the DOM sink, it must be the case that the parameter key is used to alter the control-flow of the program instead. Said another way, it implies the existence of a conditional check on the parameter key somewhere along the parameter value taint path. Thus, when producing the parameter value taint path, TRANSPARENT will also list the unique guard nodes that are present along its path. Specifically, TRANSPARENT will collect the conditional statements (e.g., if statement, ternary condition) and their required evaluated boolean value in order for said taint path to manifest. This additional information is then used to analyze the parameter key.

b) Fixed Parameter Key Analysis: To analyze the parameter key of a fixed SPA sink, TRANSPARENT relies on the fact that a parameter key is always a string. Contextualizing the guard node analysis idea with this fact, we deduce that the guard node that does a conditional check on the parameter key must be doing a string comparison. In other words, since the parameter key is always of type string and guard node statements always have a boolean value, it implies that this guard node includes an expression that compares the parameter key against a criterion and produces a boolean value.

While there are many possible string comparison criteria that return a boolean value, empirically, the SPA runtime only uses one dominant pattern to form a fixed sink: exact string matching. This kind of guard node will simply compare the property key against a constant string. If the comparison evaluates to true, then a parameter value will be assigned to a DOM sink. Empirically, this comparison has two forms:

The first is an equality expression. This is where the parameter key variable will be compared directly to a string constant with an equality comparison binary operator (e.g., ==). In this scenario, TRANSPARENT will simply deduce the string constant on the other side of the equality check as the sensitive parameter key.

The second is a membership check expression. This is where the guardian node compares the parameter key with a set of string constants. The comparison could take many forms (e.g., compounded comparison with `||` operator, function predicate), but it would still consist of exact string comparisons. This operation is typically used to test whether the property key is part of a certain namespace (e.g., SVG-related tags). In this scenario, TRANSPARENT would take the whole expression and search for all the string constants within it as the sensitive parameter key.

3) Reference SPA Sink Analysis: Reference SPA sink is different from its generic and fixed counterparts in that it does not deal with an assignment to a specific DOM sink. Instead, this kind of SPA sink works by providing a way for the component user to 'smuggle' the DOM element that is normally encapsulated by the SPA component so that it can be manipulated directly, just as if the developer were working with the native DOM API, bypassing the SPA runtime and any security mechanism that comes with it. From a component user perspective, such a DOM element is available via a 'reference' object created by the runtime, hence the name of the sink type.

The key idea to detect this type of SPA sink is to detect the flow of the encapsulated DOM element within the SPA runtime. When creating a primitive SPA component, the SPA runtime always associates it with exactly one DOM element. Thus, by tracking the flow of this particular DOM element, TRANSPARENT can figure out the path that it takes to go outside of the runtime. There are two different mechanisms in the different SPA runtimes to create a reference object: reference parameter and reference constructor.

In a reference parameter pattern, the component user provides an empty object to the render function as one of its parameters to act like a 'bucket' to which the DOM element will be assigned. After the render function has finished its initial execution, this object will be filled with the raw DOM element in a predetermined property, which then could be manipulated by the component user. To detect this pattern, TRANSPARENT searches for a dataflow path of the empty object from the render function to an assignment expression, at which the object will be placed on the left-hand side and the DOM element on the right-hand side. The existence of the path implies the existence of a 'hoist path' of the reference object from and to the outside world. This method is used by both Vue and React.

In a reference constructor pattern, the SPA framework provides the component user with a constructor function that will return the encapsulated DOM element when invoked within said component. When this component is instantiated, the constructor will search for the encapsulated DOM element within the data model and return it as the value of the constructor. To detect this pattern, TRANSPARENT also employs a form of dataflow tracking, but in contrast to the previous pattern, the DOM element acts as the taint source, and the constructor return value acts as the taint sink. The existence of this path implies the existence of an 'exit path' of the DOM element to the outside world. This method is used by Angular.

C. Static Template Mapping Analysis

The third analysis stage of TRANSPARENT is the static template mapping analysis, which figures out the specific mapping of the HTML-like template attribute name to the JavaScript-syntax framework API. The key idea is to observe the input-output pair of the transpiler in an SPA framework—i.e., a runtime module being responsible for transforming HTML templates into their corresponding JavaScript counterparts—to look for a known pattern between the attribute name and the render function parameter. These pairs could be collected from various sources. For example, we sourced them from the test suite of the transpiler, relying on the same assumption we made about the existence of a comprehensive test suite in a popular SPA runtime codebase. Specifically, within a test suite, we designate the argument of the transpiler function as the input, and the content of the `expect` statement as the output. Figure 4a shows an example of an abridged test case of a Vue transpiler.

```
1  const jsx = transpileJSX("<div domProps-innerHTML='source1'/>")
2  const sfc = transpileSFC("<div v-html='source2'/>")
3  const vnodeJSX = mockRender(jsx)
4  const vnodeSFC = mockRender(sfc)
5  expect(vnodeJSX.domProps.innerHTML).toBe('source1')
6  expect(vnodeSFC.domProps.innerHTML).toBe('source2')
```

(a) Test Case Example

```
{
    "domProps-<natveProp>": "domProps.<nativeProp>",
    "v-html": "domProps.innerHTML"
}
```

(b) Mapping Representation in JSON

Fig. 4: Example of HTML-syntax API (JSX and SFC) transpilation in a test case. All two framework API syntax would trigger the same DOM API.

TRANSPARENT receives a transpiler test suite as input. This input test suite has been filtered to only include those with a JavaScript-syntax sink in its `expect` statement. After that, it looks for an input-output pair within a single unit test that has the same attribute and property value. For example, in Figure 4a, the `expect` statement at line 5 has a `toBe` expectation of `'source1'`. TRANSPARENT parses the HTML-syntax API at lines 1 and 2 and searches for the related attribute name whose value is equal to the `expect` statement. In this case, TRANSPARENT finds the `domProps-innerHTML` attribute at line 1 also has a value of `'source1'`. From this, TRANSPARENT deduces that the HTML-syntax attribute of `domProps-innerHTML` is connected with the JavaScript-syntax parameter of `domProps.innerHTML`.

After TRANSPARENT gets the matching attribute and parameter name in a given test case, TRANSPARENT uses two different extraction methods in order to find all the HTML-syntax sinks that exist in the runtime: extrapolated mapping extraction and concrete mapping extraction.

1) Extrapolated Mapping Extraction: TRANSPARENT generalizes the found attribute name by extrapolating any detected native attribute name (e.g., `src`) and native property name (e.g., `innerHTML`) into `<nativeAttr>` and `<nativeProp>`, respectively. In the specific example of Figure 4a, the `domProps-innerHTML` attribute (line 1) is mapped into an object in the JavaScript-syntax sink (line 5). This HTML-syntax sink applies to every `domProps-<nativeProp>`

attribute, but is only being tested for `innerHTML` specifically, hence the need for extrapolation.

To properly extrapolate the attribute name in the test case, TRANSPARENT considers two extrapolation rules to account for the difference in syntax rules between HTML and JavaScript. The first rule is that TRANSPARENT ignores characters that are illegal in JavaScript variable names (e.g., the hyphen character `-`), and the second rule is that TRANSPARENT ignores any miscellaneous prefix in the attribute name (e.g., ignoring `bind-` in `bind-src`).

2) Concrete Mapping Extraction: TRANSPARENT directly maps the found attribute name to the render function parameter. Concrete mapping is done if TRANSPARENT does not find any sensitive native attribute or property name in the found attribute name. For example, this is what happened at line 2 in Figure 4a since `v-html` does not match any sensitive native attribute or property name. Once TRANSPARENT sees such a test case, TRANSPARENT deduces that `v-html` is mapped to `domProps.innerHTML`, without any generalization.

The output of both of these extractions is a set of mappings from the HTML-syntax sink to the corresponding JavaScript-syntax sink, as shown in Figure 4b. TRANSPARENT then gets the sensitive attribute name in HTML-syntax sinks by getting the keys of this mapping set.

V. EVALUATION

In this section, we evaluate our design and detail the results of TRANSPARENT when applied to each SPA framework and the application built on top of them.

A. Evaluation Setup

Our implementation of TRANSPARENT consists of TypeScript code and CodeQL queries. We developed a language binding between TypeScript and the CodeQL command-line interface in order for TRANSPARENT to run templated CodeQL queries and parse their results, which are formatted in `bqrs`. We use CodeQL version 2.21.1 with version 2.6.1 of the JavaScript and TypeScript standard library.

For comparison with the vanilla CodeQL (built-in queries), we use security-related SPA sinks that are already included in this library. Specifically, we use CodeQL classes related to client-side taint-style vulnerabilities such as DOM-based XSS, client-side URL redirection, and code injection. This includes both classes that are specified in the dataflow modules and classes that are specified in the SPA framework modules.

For our target frameworks, we analyzed Angular v17, React v18.2, and Vue v2.7.16. We chose these specific versions because they have the most mature supported ecosystem as of fall 2024.

For our evaluation, we composed two datasets: ① GitHub repositories dataset, and ② known vulnerabilities dataset.

The first dataset is composed of GitHub repositories of SPAs that have the analyzed target framework in their `package.json` file. We then clone and run TRANSPARENT against them to detect potential vulnerabilities. This dataset is used to evaluate TRANSPARENT's false positive rate and zero-day vulnerabilities search. The second dataset is composed of SPA repositories that are known to be exploitable. We compose this dataset from the public CVE list as well as GitHub Security Advisory for client-side taint-style vulnerabilities. This dataset is used to evaluate TRANSPARENT's false negative rate.

We ran all of our experiments on an Ubuntu 24.04 LTS server equipped with Intel Xeon E5-2620v4 and 64 GiB of RAM.

We did our evaluation in relation to the following research questions:

- **RQ1**: How accurate is TRANSPARENT in terms of false positives and false negatives?
- **RQ2**: How effective is TRANSPARENT in uncovering zero-day vulnerabilities?
- **RQ3**: What SPA sinks does TRANSPARENT find?
- **RQ4**: How does TRANSPARENT perform in terms of computational overhead?

B. RQ1: Accuracy

We evaluate TRANSPARENT based on its accuracy in detecting vulnerabilities in SPA components. This evaluation is split into two ways: false negative rate (FNR) and false discovery rate (FDR). Table IV summarizes this evaluation.

1) False negative rate: Ideally, a vulnerability detection tool should report all vulnerabilities in the dataset. False negative rate measures the degree to which said detection tool fails to report vulnerabilities. We evaluate FNR using the known vulnerabilities dataset.

To evaluate FNR on the known vulnerabilities dataset, we gather a list of publicly acknowledged vulnerabilities (CVE and GitHub Security Advisory) that are related to the framework runtime and evaluate TRANSPARENT's ability to detect the vulnerable path. We gathered a total of 56 public vulnerabilities and evaluated them against TRANSPARENT and Vanilla CodeQL. If the exploit detail mentions a third-party library being the culprit, we also include it in the analysis. In a vulnerability that involves a server (e.g., stored XSS), we consider both tools to successfully detect the vulnerability if they can show a path from a request's response (e.g., fetch, URL).

Among those 56, TRANSPARENT fails to detect 11, and vanilla CodeQL fails to detect 35. Looking at the root cause of this outperformance, we found that it is caused by TRANSPARENT's expanded set of SPA sinks. In particular, TRANSPARENT detects many sinks that are written in Vue JavaScript-syntax SPA sinks (domProps.`<nativeProp>`), which vanilla CodeQL fails to detect. This syntax is particularly common in some component libraries. On the other hand, vanilla CodeQL and TRANSPARENT share the same set of false negatives, which is mainly caused by incomplete in-component dataflow, which is the limitation of the underlying static analysis engine.

2) False discovery rate: Ideally, a vulnerability detection tool should only report vulnerabilities. False discovery rate measures the degree to which said detection tool reports non-vulnerabilities. We evaluate FDR using the GitHub repositories dataset.

TABLE III: List of zero-day vulnerabilities that use our newly-found sinks. Ack (ext. sanitizer) means that the developer acknowledged that it is exploitable, but sanitization duty falls into the responsibility of the library user.

GitHub Repository Name	Stargazer	Framework	Sinks	Syntax	Vuln. Type	Status
the1812/Bilibili-Evolved	23k+	Vue	domProps.`<nativeProp>`	JavaScript	XSS	Fixed
apostrophe/apostrophe	4k+	Vue	ref	JavaScript	CSV injection	Reported
bootstrap-vue/bootstrap-vue	14.5k+	Vue	domProps.`<nativeProp>`	JavaScript	XSS	Reported
alfonsobries/vue-tailwind	2.2k+	Vue	domProps.`<nativeProp>`	JavaScript	XSS	Reported
miaolz123/vue-markdown	1.9k+	Vue	domProps.`<nativeProp>`	JavaScript	XSS	Reported
iview/iview	24k+	Vue	domProps.`<nativeProp>`	JavaScript	XSS	Reported
JosephusPaye/Keen-UI	4.1k+	Vue	domProps.`<nativeProp>`	JavaScript	XSS	Ack (ext. sanitizer)
jiangshanmeta/vue-admin	170+	Vue	domProps.`<nativeProp>`	JavaScript	XSS	Reported
surveyjs/survey-library	3.8k+	React	xlinkHref	HTML (JSX)	XSS	Ack (ext. sanitizer)
salesforce/design-system-react	900+	React	xlinkHref	HTML (JSX)	XSS	Acknowledged
evaletolab/ng2-markdown	1+	Angular	ref	JavaScript	XSS	Reported

To evaluate FDR on the GitHub repositories dataset, we run both TRANSPARENT and vanilla CodeQL on each repository in the dataset and manually test the reported vulnerable path(s) within those repositories to check whether they are exploitable. In contrast to our zero-day vulnerability evaluation, we also count applications whose SPA sinks are included in the vanilla CodeQL. In a repository that has a lot of overlapping paths (e.g., component libraries where there are compounded components), we only count the inner-most paths due to the abundance of potential paths. This effectively counts all paths that have the same DOM sink location as one vulnerable path.

Out of the 877 repositories in the GitHub dataset, TRANSPARENT reports 571 vulnerable paths, and vanilla CodeQL reports 340 vulnerable paths. Due to the impracticality of manually checking them for exploitability, given the time constraint, we randomly sample about 10% from each tool's reported vulnerable paths to check for exploitability.

The result is that TRANSPARENT has a comparable FDR value to CodeQL. This is because TRANSPARENT and CodeQL share the same underlying engine, making it suffer from similar issues related to overtainting. After analyzing the non-vulnerable paths that are marked as vulnerable, the three most common causes are as follows: ① Non-controllable taint sources (e.g., safe server endpoints). ② Unidentified sanitization logic (e.g., based on a regular expression). ③ Inclusion of auxiliary code (e.g., inclusion of minified code in the repository). We will discuss their potential remedies in the discussion section (§VI).

C. RQ2: Effectiveness in Finding Zero-Day Vulnerabilities

We also evaluate TRANSPARENT on its capability of detecting vulnerabilities in SPA components that run on the top 3 most popular SPA runtimes: Vue, React, and Angular. To do this, we form the GitHub dataset where we crawled repositories from GitHub for SPAs that are built on top of frameworks targeted by TRANSPARENT. We limit our crawling to repositories that have at least 1 GitHub stargazer due to the number of crawled repositories. This process results in 877 total of repositories. We then run TRANSPARENT on each framework runtime, then analyze the crawled SPA components. Of those SPA components that are detected to have SPA sinks, we manually examine repositories that have SPA sinks that only TRANSPARENT could detect (i.e., not included in the vanilla CodeQL) for an exploit.

TABLE IV: Accuracy breakdown of TRANSPARENT compared to baseline

Tool	FNR	FDR
TRANSPARENT	11/56 (19.6%)	24/57 (42.1%)
Vanilla CodeQL	35/56 (62.5%)	17/34 (50.0%)

Table III summarizes the zero-day vulnerabilities that TRANSPARENT has detected across all SPA runtimes. The "Sinks" column represents the type of SPA sinks that the vulnerability uses. Vulnerabilities that are marked with "Acknowledged (external sanitizer)" are usually handled by the developer by revising the documentation to explicitly warn/remind users of an external sanitizer, highlighting the impacts of our findings. Vulnerabilities marked with "Fixed" are patched by their developers. The rest with "Reported" status are validated as exploitable with our manual analysis and reported to their developers, but we have not received any feedback at the time of the paper writing.

D. RQ3: Found SPA Sinks

From our false negative rate evaluation, we found that the primary cause that contributes to TRANSPARENT's outperformance is its expanded set of SPA sinks. We take a look further at the novel sinks produced by TRANSPARENT to understand its significance. Table V lists all the SPA sinks that TRANSPARENT has discovered. While these sinks are from all frameworks, they are not equally distributed. This distribution is mostly a function of the amount of syntax a framework supports and the presence of dataflow sanitization inside the runtime. Most notably, Angular only has one new sink. This is due to the fact that Angular only has one syntax (HTML) and implements a comprehensive sanitization mechanism called SafeType[43] that forbids most dataflows from being exploitable. Interestingly, we found that React also has a sanitization mechanism, but it is disabled by default. Vue supports three syntaxes and has no internal sanitization mechanism, which explains why Vue has the largest amount of novel sinks.

1) Novel Sinks: We compare SPA sinks discovered by TRANSPARENT with those used by prior works or those with security warnings in the official documentation of the SPA. In total, TRANSPARENT produced 19, of which 14 are

TABLE V: List of sensitive framework API. `nativeAttr` and `nativeProps` correspond to any attributes and properties respectively of the underlying HTML tag. For list of sensitive attributes and properties, refer to Table II.

Sensitive Framework API	Framework	Syntax	Vanilla CodeQL	ReactAppScan	Warning	TRANSPARENT
attrs.<nativeAttr>	Vue	JavaScript-syntax	No	No	No	Yes
domProps.<nativeProp>	Vue	JavaScript-syntax	No	No	Yes [45]	Yes
ref	Vue	JavaScript-syntax	No	No	No	Yes
<nativeAttr>	Vue	HTML-syntax (JSX)	No	No	No	Yes
attrs-<nativeAttr>	Vue	HTML-syntax (JSX)	No	No	No	Yes
domProps-<nativeProp>	Vue	HTML-syntax (JSX)	No	No	No	Yes
domProps<nativeProp>	Vue	HTML-syntax (JSX)	No	No	Yes [45]	Yes
ref	Vue	HTML-syntax (JSX)	No	No	No	Yes
<nativeAttr>	Vue	HTML-syntax (SFC)	Some	No	No	Yes
v-html	Vue	HTML-syntax (SFC)	Yes	No	Yes [45]	Yes
ref	Vue	HTML-syntax (SFC)	No	No	No	Yes
<nativeAttr>	React	JavaScript-syntax	No	No	No	Yes
dangerouslySetInnerHtml	React	JavaScript-syntax	No	No	No	Yes
ref	React	JavaScript-syntax	No	No	No	Yes
<nativeAttr>	React	HTML-syntax (JSX)	Some	No	No	Yes
dangerouslySetInnerHtml	React	HTML-syntax (JSX)	Yes	Yes	Yes [38]	Yes
ref	React	HTML-syntax (JSX)	No	Yes	No	Yes
renderer2.setProperty	Angular	JavaScript-syntax	Yes	No	No	Yes
ref	Angular	JavaScript-syntax	No	No	Yes [44]	Yes

not listed by the benchmark Vanilla CodeQL. Furthermore, after a careful examination of prior works, we find that only ReactAppScan [21] adopts React's JSX `ref` in addition to a well-known sink (`dangerouslySetInnerHTML`). None of the other four React sinks, 11 Vue sinks, or two Angular sinks are considered by other prior works.

Meanwhile, we also examined the official documentation of each SPA. While all of these sinks are documented, we found that not all of them came with a security warning. Among these sinks, we found that only `domProps` sinks in Vue [45], `dangerouslySetInnerHtml` JSX sink in React [38], and `ref` sink in Angular [44] come with an explicit vulnerability warning, whereas other sinks do not. In summary, 10 sinks discovered by TRANSPARENT are neither used by prior works nor documented with security warnings.

2) Ablation Studies: We perform ablation studies to determine the impact of TRANSPARENT's components by removing them and observing the number of SPA sinks found. Specifically, we remove the autostitch, the extrapolated mapping, and the concrete mapping from TRANSPARENT and call them TRANSPARENT-no-autostitch, TRANSPARENT-no-extrapol-mapping, and TRANSPARENT-no-concrete-mapping.

Table VI summarizes our experiment findings. TRANSPARENT-no-autostitch does not find any SPA sinks due to an incomplete call graph. TRANSPARENT-no-extrapol-mapping managed to find 12/19 sinks, failing to discover Vue's and React's generic SPA sinks as well as React's `dangerouslySetInnerHTML`, because of the lack of mapping between HTML and JavaScript syntaxes. Additionally, TRANSPARENT-no-concrete-mapping managed to find 14/19 sinks, failing to discover Vue's `v-html` sink and all reference sinks.

E. RQ4: Computational Overhead

The final research question is about the runtime overhead of the analysis. We split this evaluation into two: how long does TRANSPARENT analyze each runtime, and how long does TRANSPARENT analyze applications? To contextualize the duration, we also provide the line-of-code count for the runtime and applications. We use the `cloc`[8] tool v2.02 to accomplish this. For the SPA runtimes, we count the amount of TypeScript and JavaScript code. For the applications themselves, we count the amount of TypeScript, JavaScript,

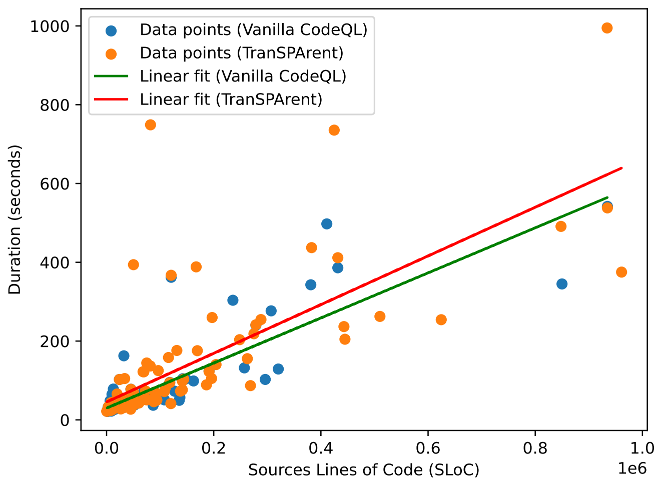


Fig. 5: Analysis performance overhead of 100 randomly sampled applications with respect to their line-of-code count.

TABLE VI: Ablation Study for Found SPA Sinks

Variant	Found Sinks
TRANSPARENT-no-autostitch	0/19
TRANSPARENT-no-extrapol-mapping	12/19
TRANSPARENT-no-concrete-mapping	14/19
TRANSPARENT	19/19

TABLE VII: Runtime overhead of framework abstraction

Runtime	Analysis Time	LoC
Vue	57m	73k
React	1h 12m	353k
Angular	1h 15m	659k

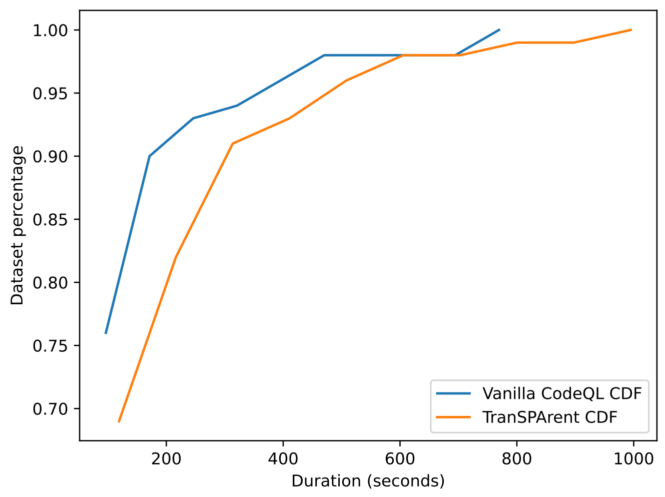


Fig. 6: CDF of analysis performance overhead of 100 randomly sampled applications.

HTML, JSX, and Vue SFC code.

1) Runtime Analysis Overhead: We evaluate the time it took for TRANSPARENT to abstract each SPA runtime in Table VII. For all three of the SPA runtimes under test, the time it takes to analyze each of them falls around one hour. The analysis time scales well with respect to the line-of-code count since CodeQL also performs optimization under the hood to execute each analysis query. Because each framework only needs to be processed once, this time will effectively be amortized as more applications get analyzed.

2) Application Analysis Overhead: We randomly sample 100 applications from our GitHub repositories dataset and plot their analysis time with respect to the line-of-code count in Figure 5. Our evaluation shows that TRANSPARENT's analysis time grows linearly as the size of the application grows, and the difference in trend line is negligible compared to the vanilla CodeQL (0.62 seconds / kloc and 0.57 seconds / kloc, respectively).

Figure 6 shows the cumulative distribution function of the analysis performance overhead. It shows that all of the sampled applications finish their analysis in minutes. In particular, around 90% of applications are successfully analyzed within 5 minutes. This is a negligible difference in comparison to the baseline, where 95% of applications are analyzed under 5 minutes.

These results show that TRANSPARENT's high degree of efficiency when processing SPAs, from the SPA runtime analysis up to the application itself.

VI. Discussion and Future Works

This section will briefly articulate common discussion points that we had during the research process.

Event handlers. One class of taint-style vulnerabilities in native HTML that we do not take into account in this paper is event handlers. This is due to the fact that event handlers in SPA are usually provided as a native JavaScript function instead of an evaluated string. This is unlikely to be manipulable by an end user; thus, we do not consider it a threat in the context of an SPA framework.

Generalizability for other SPA frameworks and versions. We have demonstrated the generalizability of our methods by applying them to three different SPA frameworks and discovering hidden SPA sinks within them. The idea of TRANSPARENT is general and could be extended to other SPA frameworks because we designed it based on the general architecture of multiple single-page application frameworks. Specifically, the idea that an SPA component encapsulates a DOM component is general across different frameworks, and different architectural implementations of them are handled by different kinds of analysis within TRANSPARENT (e.g., generic, fixed, and reference SPA sink analysis).

While designing TRANSPARENT, we also tried to analyze different versions of the same SPA framework. Our empirical finding shows that, in practice, different versions of SPA frameworks have tried to maintain a stable and backward-compatible API. The result is that sinks that are discovered in one version of an SPA framework could potentially be applied to another version, making analyzing individual versions of each SPA framework of limited utility. While this is not a guarantee, other static analysis tool that supports analyzing SPAs like CodeQL also have a similar approach, providing only one module per framework, instead of one module per framework version.

Prerequisites for running TRANSPARENT. There are two conditions for TRANSPARENT's analysis. First, TRANSPARENT needs a working test suite against the SPA framework. This is needed for the autostitch process to successfully augment the call graph of a commercial-off-the-shelf tool. In our experiment, we find specifically that TRANSPARENT obtains the greatest benefit from test suites that extensively test the end-to-end rendering pipeline. Empirically, since TRANSPARENT only needs a test suite from the SPA *frameworks* (and not the applications themselves), we find that the test suite it uses is always comprehensive and has high coverage (thus always tests the end-to-end rendering pipeline), given their popularity.

Second, TRANSPARENT needs explicit type information, since some parts of TRANSPARENT's analyses rely on it. Fortunately, most SPA frameworks today (as is the case with many large JavaScript projects in general) are written in typed languages, such as TypeScript.

Static site generation and server-side rendering. Since SPAs only start with one HTML page and dynamically generate content on top, SPAs generally do not have much static content to be discovered by search engines. This downside leads many SPA frameworks to develop ways to offset some of their rendering to the server, either via Static Site Generation (SSG), where each page is rendered into HTML at build-time, or Server-Side Rendering (SSR), where some components are rendered into HTML at request-time.

However, they will eventually need to be controlled by the SPA runtime on the client (typically in a process called "hydration") and could have the same SPA sink vulnerabilities as their purely client-side SPA. We consider this kind of application to be out of scope since it also necessitates dataflow tracking on the server. Expanding the automated framework abstraction approach to cover not just SPA but also SSG and SSR applications is also a promising avenue for future work.

Sanitization responsibility. While performing our evaluation on GitHub repositories, we found that many of them reside in component libraries. When we disclose them to the library developer, one common response is that sanitization should be the responsibility of the library user instead. This sparks a discussion about the proper location of the sanitization code. We think it is prudent to implement sanitization code at the SPA runtime level. Out of the three SPA runtimes we examined, only Angular implements sanitization by default (as part of its SafeType[43] scheme), and we found the least amount of vulnerable applications built on top of it.

VII. Related Works

This section will briefly contextualize our work by comparing it to other related works within the literature.

SPA framework analysis. The security aspect of SPA frameworks is relatively unexplored within the web security literature. To the best of our knowledge, ReactAppScan [21] is the only related work in this area. However, ReactAppScan focuses solely on the in-component dataflow instead of incorporating in-framework dataflow, which is orthogonal to our work. ReactAppScan, as its name suggests, also only focuses on React, whereas our work is trying to develop a generic method for analyzing arbitrary SPAs. SPA-specific threats are also mentioned in the work of Felsch et al. [15], which in itself is not a work on SPA vulnerabilities, but recognizes the concept of SPA-specific vulnerabilities (in particular, Angular sandbox escape).

Aside from academic literature, commercial-off-the-shelf static analysis tools like CodeQL [9], SonarQube [4], and Fortify SCA [16] do have the capability of analyzing SPA. However, many of them focus more on analyzing code cleanliness. If they have security-related capabilities at all, they have similar limitations of not incorporating in-framework dataflow. Moreover, most of them require a license to unlock the SPA security-related dataflow analysis capability, CodeQL being the notable exception.

JavaScript runtime framework analysis. JavaScript-based frameworks are not limited only to SPA, and there have been several works that focus on them.

One example of this is a web-desktop hybrid app framework, such as Electron [11]. The work of Jin et al. [23] and XGuard [47] focuses on preventing injection vulnerabilities in the host system. There is also a web-mobile hybrid app framework like Apache Cordova [10] (formerly PhoneGap), on which works like NoFrak [17] based its injection attacks studies upon. There is also a trend of mobile superapp security, such as the work of Zhang et al. [50].

Other common JavaScript frameworks include browser extensions and web servers. For example, CoCo [49] tries to detect vulnerabilities in browser extensions through an abstract interpretation technique. On the other hand, Arcanum [48] and Mystique [7] focus on dealing with privacy-sensitive information that browser extensions might upload to the cloud. For another example, TEFuzz [51] focuses on preventing Server-Side Template Injection threats through a fuzzing process. Whereas the work of Squicina et al. [39] focuses more on insecure cookie handling in various server frameworks.

All of these works focus on specific vulnerabilities, threat models, and the structure of the runtime and thus have a different scope from our work. However, TRANSPARENT does fit in the broader category of JavaScript runtime framework analysis.

JavaScript taint-style vulnerability analysis. A substantial body of work exists within the literature regarding the detection of JavaScript taint-style vulnerabilities with various techniques to deal with its dynamic nature.

One popular method is abstract interpretation. ODGen [29], ObjLupAnsys [28], and Nodest [33] use abstract interpretation for analyzing taint-style vulnerabilities in Node.js packages. Deemon [36] and JAW [25] use abstract interpretation to detect CSRF vulnerabilities. There are also tools [40], [26], [35] that do taint-style vulnerability detection without abstract interpretation. However, none of these tools address SPA-specific problems, such as the multiple syntax problem. There are also tools [6] that focus on application-specific missing edge diagnostic tools for JavaScript applications, but do not focus on vulnerability detection like TRANSPARENT.

Web client security. We also acknowledge the vast body of work on detecting and mitigating common web client vulnerability classes in general.

One common web client vulnerability that appears a lot in this paper is XSS [41], [31]. A classic work in this topic is Document Structure Integrity [32] that enforces the integrity of trusted code to prevent tampering from untrusted input. Another approach is done by Lekies et al. [27], who use a dynamic analysis method to track the flow of untrusted data paired with an exploit generation system. There are also works focusing on CSRF [25], [36], browser tracking and fingerprinting [46], [34], [19], and JavaScript malware [12], [5], [14], [13].

These works, however, mainly target framework-less web applications. TRANSPARENT contributes in this dimension in that it detects client-side taint-style vulnerabilities even in the presence of the ever-ubiquitous SPA runtime.

VIII. Conclusion

In this paper, we introduced TRANSPARENT, a novel and generic approach to detect taint-style vulnerabilities in SPAs by doing automated framework abstraction. TRANSPARENT uses static taint path analysis in order to figure out how the parameter of a native framework component could flow into a sensitive DOM API. To do this, TRANSPARENT also uses a dynamic analysis technique called 'autostitch' to augment missing call edges within the static analysis part. Finally, to discover framework sinks that are not written in a JavaScript syntax (i.e., HTML-syntax sinks), TRANSPARENT uses a static parameter mapping analysis to reveal them, given a corresponding JavaScript-syntax sink. Our evaluation shows that TRANSPARENT revealed 14 new framework sinks, which are used by 11 zero-day vulnerabilities. TRANSPARENT reveals more framework sinks compared to those that are used by CodeQL standard queries, which is the state-of-the-art static analysis tool for SPA vulnerability detection.

ACKNOWLEDGMENT

We would like to thank anonymous reviewers and the shepherd for their helpful comments and feedback. This work was supported in part by the National Science Foundation (NSF) under grants CNS-21-54404 and CNS-20-46361. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of NSF.

ETHICS CONSIDERATIONS

We see two potential topics of ethical discussion that could stem from our work.

The first is regarding the experiment's impact on live systems. Due to the client-side and repository-centered nature of our experiments, the majority of our experiment cases do not need a live system to begin with, since they could be run locally. However, there are notable exceptions for cases such as reflected and stored XSS, where the server source code is not available fully and cannot be deployed locally. In such cases, we make sure that our experiment does not interfere with the functionality of the system with respect to all the stakeholders involved, which are the user and the administrator of said system. We make sure that the impact of the experiment done in such a system is only limited to our client and does not have a system-wide effect.

The second is regarding responsible vulnerability disclosure. We report all zero-day vulnerabilities that we found and report them to the developer according to their security policy on GitHub, and also notify them that we plan to include the vulnerability in an academic publication no earlier than 45 days after the disclosure to allow ample time for fixes. If the developer does not have a security policy, we opened a GitHub pull request with a simple sanitization fix or emailed the developer regarding the vulnerability. The "Status" column on Table III refers to the acknowledgment status by the developer. For library-level vulnerabilities specifically, an "ext. sanitizer" status means that the developer of the library acknowledges that their library could be used in an exploit, but the sanitization process is meant to be done externally (i.e., by the library user), instead of within the library. For the sake of thoroughness, we also contacted the developer of each framework regarding our findings.

REFERENCES

[1] Angular. Retrieved 1/20, 2025 from https://angular.io/.

[2] SPA: Single Page Application. Retrieved 1/20, 2025 https://developer.mozilla.org/en-us/docs/glossary/spa.

[3] PWA: Progressive Web Apps. Retrieved 1/20, 2025 from https://developer.mozilla.org/en-us/docs/web/progressive_web_apps.

[4] SonarQube by SonarSource. Retrieved 5/20, 2025 https://www.sonarsource.com/products/sonarqube/.

[5] Yinzhi Cao, Xiang Pan, Yan Chen, and Jianwei Zhuge. Jshield: Towards real-time and vulnerability-based detection of polluted drive-by download attacks. In *Proceedings of the 30th Annual Computer Security Applications Conference*, pages 466–475, 2014.

[6] Madhurima Chakraborty, Renzo Olivares, Manu Sridharan, and Behnaz Hassanshahi. Automatic root cause quantification for missing edges in javascript call graphs. In *36th European Conference on Object-Oriented Programming (ECOOP 2022)*, pages 3–1. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2022.

[7] Quan Chen and Alexandros Kapravelos. Mystique: Uncovering information leakage from browser extensions. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, page 1687–1700, New York, NY, USA, 2018. Association for Computing Machinery.

[8] cloc. Retrieved 1/20, 2025 https://github.com/aldanial/cloc.

[9] GitHub CodeQL. Retrieved 1/20, 2025 from https://codeql.github.com/.

[10] Apache Cordova. Retrieved 1/20, 2025 from https://cordova.apache.org/.

[11] Electron. Retrieved 1/20, 2025 from https://www.electronjs.org/.

[12] Aurore Fass, Michael Backes, and Ben Stock. Hidenoseek: Camouflaging malicious javascript in benign asts. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 1899–1913, 2019.

[13] Aurore Fass, Michael Backes, and Ben Stock. Jstap: a static pre-filter for malicious javascript detection. In *Proceedings of the 35th Annual Computer Security Applications Conference*, pages 257–269, 2019.

[14] Aurore Fass, Robert P Krawczyk, Michael Backes, and Ben Stock. Jast: Fully syntactic detection of malicious (obfuscated) javascript. In *Detection of Intrusions and Malware, and Vulnerability Assessment: 15th International Conference, DIMVA 2018, Saclay, France, June 28–29, 2018, Proceedings 15*, pages 303–325. Springer, 2018.

[15] Dennis Felsch, Mario Heiderich, Frederic Schulz, and Jörg Schwenk. How private is your private cloud? security analysis of cloud control interfaces. In *Proceedings of the 2015 ACM Workshop on Cloud Computing Security Workshop*, CCSW '15, page 5–16, New York, NY, USA, 2015. Association for Computing Machinery.

[16] OpenText SCA (Fortify). Retrieved 5/20, 2025 https://www.opentext.com/products/static-application-security-testing.

[17] Martin Georgiev, Suman Jana, and Vitaly Shmatikov. Breaking and fixing origin-based access control in hybrid web/mobile application frameworks. In *Proc. of the Network and Distributed System Security Symposium (NDSS'14)*, volume 2014, page 1, 2014.

[18] BiliBili GitHub. Retrieved 1/20, 2025 https://github.com/the1812/bilibili-evolved.

[19] Alejandro Gómez-Boix, Pierre Laperdrix, and Benoit Baudry. Hiding in the crowd: an analysis of the effectiveness of browser fingerprinting at large scale. In *Proceedings of the 2018 world wide web conference*, pages 309–318, 2018.

[20] Salvatore Guarnieri. {GULFSTREAM}: Staged static analysis for streaming {JavaScript} applications. In *USENIX Conference on Web Application Development (WebApps 10)*, 2010.

[21] Zhiyong Guo, Mingqing Kang, VN Venkatakrishnan, Rigel Gjomemo, and Yinzhi Cao. Reactappscan: Mining react application vulnerabilities via component graph. In *S&P*, 2024.

[22] Behnaz Hassanshahi, Hyunjun Lee, and Paddy Krishnan. Gelato: Feedback-driven and guided security analysis of client-side web applications. In *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 618–629. IEEE, 2022.

[23] Zihao Jin, Shuo Chen, Yang Chen, Haixin Duan, Jianjun Chen, and Jianping Wu. A security study about electron applications and a programming methodology to tame dom functionalities. In *NDSS*, 2023.

[24] BiliBili-Evolved jsdelivr statistics. Retrieved 1/20, 2025 https://www.jsdelivr.com/package/gh/the1812/bilibili-evolved.

[25] Soheil Khodayari and Giancarlo Pellegrino. JAW: Studying client-side CSRF with hybrid property graphs and declarative traversals. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 2525–2542. USENIX Association, August 2021.

[26] Pierre Laperdrix, Oleksii Starov, Quan Chen, Alexandros Kapravelos, and Nick Nikiforakis. Fingerprinting in style: Detecting browser extensions via injected style sheets. In *30th USENIX Security Symposium*

(USENIX Security 21), pages 2507–2524. USENIX Association, August 2021.

[27] Sebastian Lekies, Ben Stock, and Martin Johns. 25 million flows later: large-scale detection of dom-based xss. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 1193–1204, 2013.

[28] Song Li, Mingqing Kang, Jianwei Hou, and Yinzhi Cao. Detecting node.js prototype pollution vulnerabilities via object lookup analysis. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2021, page 268–279, New York, NY, USA, 2021. Association for Computing Machinery.

[29] Song Li, Mingqing Kang, Jianwei Hou, and Yinzhi Cao. Mining node.js vulnerabilities via object dependence graph and query. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 143–160, Boston, MA, August 2022. USENIX Association.

[30] Magnus Madsen, Benjamin Livshits, and Michael Fanning. Practical static analysis of javascript applications in the presence of frameworks and libraries. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 499–509, 2013.

[31] William Melicher, Anupam Das, Mahmood Sharif, Lujo Bauer, and Limin Jia. Riding out domsday: Towards detecting and preventing dom cross-site scripting. In *2018 Network and Distributed System Security Symposium (NDSS)*, 2018.

[32] Yacin Nadji, Prateek Saxena, and Dawn Song. Document structure integrity: A robust basis for cross-site scripting defense. In *NDSS*, volume 20, 2009.

[33] Benjamin Barslev Nielsen, Behnaz Hassanshahi, and François Gauthier. Nodest: feedback-driven static analysis of node.js applications. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2019, page 455–465, New York, NY, USA, 2019. Association for Computing Machinery.

[34] Xiang Pan, Yinzhi Cao, and Yan Chen. I do not know what you visited last summer: Protecting users from third-party web tracking with trackingfree browser. In *Proceedings of the 2015 Annual Network and Distributed System Security Symposium (NDSS), San Diego, CA*, 2015.

[35] Joonyoung Park, Jihyeok Park, Dongjun Youn, and Sukyoung Ryu. Accelerating javascript static analysis via dynamic shortcuts. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1129–1140, 2021.

[36] Giancarlo Pellegrino, Martin Johns, Simon Koch, Michael Backes, and Christian Rossow. Deemon: Detecting csrf with dynamic analysis and property graphs. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, page 1757–1771, New York, NY, USA, 2017. Association for Computing Machinery.

[37] React. Retrieved 1/20, 2025 from https://react.dev/.

[38] Dangerously setting the inner HTML. Retrieved 1/20, 2025 from https://react.dev/reference/react-dom/components/common#dangerously-setting-the-inner-html.

[39] Marco Squarcina, Pedro Adão, Lorenzo Veronese, and Matteo Maffei. Cookie crumbles: Breaking and fixing web session integrity. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 5539–5556, Anaheim, CA, August 2023. USENIX Association.

[40] Cristian-Alexandru Staicu, Michael Pradel, and Ben Livshits. Understanding and automatically preventing injection attacks on node. js. In *Network and Distributed System Security Symposium (NDSS)*, 2018.

[41] Marius Steffens, Christian Rossow, Martin Johns, and Ben Stock. Don't trust the locals: Investigating the prevalence of persistent client-side cross-site scripting in the wild. *CISPA*, 2019.

[42] Vue. Retrieved 1/20, 2025 from https://vuejs.org/.

[43] Pei Wang, Julian Bangert, and Christoph Kern. If it's not secure, it should not compile: Preventing dom-based xss in large-scale web development with api hardening. In *Proceedings of the 43rd International Conference on Software Engineering*, ICSE '21, page 1360–1372. IEEE Press, 2021.

[44] Angular 17 ElementRef Security Warning. Retrieved 11/26, 2025 https://v17.angular.io/api/core/elementref.

[45] Vue 2 Security Warning. Retrieved 5/20, 2025 https://v2.vuejs.org/v2/guide/security.

[46] Shujiang Wu, Pengfei Sun, Yao Zhao, and Yinzhi Cao. Him of many faces: Characterizing billion-scale adversarial and benign browser fingerprints on commercial websites. In *NDSS*, 2023.

[47] Feng Xiao, Zheng Yang, Joey Allen, Guangliang Yang, Grant Williams, and Wenke Lee. Understanding and mitigating remote code execution vulnerabilities in cross-platform ecosystem. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, pages 2975–2988, 2022.

[48] Qinge Xie, Manoj Vignesh Kasi Murali, Paul Pearce, and Frank Li. Arcanum: Detecting and evaluating the privacy risks of browser extensions on web pages and web content. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 4607–4624, Philadelphia, PA, August 2024. USENIX Association.

[49] Jianjia Yu, Song Li, Junmin Zhu, and Yinzhi Cao. Coco: Efficient browser extension vulnerability detection via coverage-guided, concurrent abstract interpretation. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, CCS '23, page 2441–2455, New York, NY, USA, 2023. Association for Computing Machinery.

[50] Yue Zhang, Yuqing Yang, and Zhiqiang Lin. Don't leak your keys: Understanding, measuring, and exploiting the appsecret leaks in mini-programs. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, CCS '23, page 2411–2425, New York, NY, USA, 2023. Association for Computing Machinery.

[51] Yudi Zhao, Yuan Zhang, and Min Yang. Remote code execution from SSTI in the sandbox: Automatically detecting and exploiting template escape bugs. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 3691–3708, Anaheim, CA, August 2023. USENIX Association.

Appendix A
Artifact Appendix

This is the artifact that accompanies the TRANSPARENT paper, an SPA vulnerability detection tool that automatically abstracts SPA frameworks using a combination of static and dynamic analysis to reveal framework-specific sinks. TRANSPARENT works in two broad steps: first, it analyzes the SPA runtime source code to abstract its SPA-specific sinks, which is done once per framework. Secondly, it uses the resulting SPA-specific sinks to look for vulnerabilities in SPA applications.

This artifact provides the source code and datasets used to reproduce the two main evaluation results, namely the list of intermediate SPA-specific sinks and the accuracy table over the known vulnerability dataset and the positive label GitHub dataset.

A. Description & Requirements

1) How to access: The artifacts are available permanently (through Zenodo) and accessible through GitHub at the following links:

- https://doi.org/10.5281/zenodo.17822391
- https://github.com/diwangs/transparent-ae

2) Hardware dependencies: The artifact can be run on commodity hardware. We ran our experiment twice on two different hardware with identical key results: ① on a server equipped with an Intel Xeon E5-2620v4 CPU (8 cores, 2.1 GHz) paired with 64 GB of RAM, and ② on a laptop equipped with an AMD Ryzen 7 7840U CPU (8 cores, 3.3 GHz) paired with 16 GB of RAM. The runtime estimates in this appendix will be based on the latter, as its hardware is more approachable.

3) Software dependencies: We require a Linux machine with two software dependencies: ① **Nix package manager** to install all the rest of the software dependencies, and ② **Git LFS** to properly download the required datasets from GitHub.

4) Benchmarks: We provide two datasets to be used as benchmark:

- **Known vulnerabilities dataset** - This dataset is composed of SPA repositories that are known to be exploitable. We compose this dataset from the public CVE list as well as GitHub Security Advisory (GHSA) for client-side taint-style vulnerabilities, amounting to 56 public vulnerabilities. This dataset is used to evaluate TranSPArent's false negative rate and is located at the `accuracy/fnr/` directory
- **Positive label GitHub dataset** - This dataset is composed of GitHub repositories of SPAs that are marked as vulnerable by TRANSPARENT. In the paper, we sample 10% of alerts that are produced by TRANSPARENT and its baseline, vanilla CodeQL, amounting to 57 and 24 alerts, respectively. This dataset is used to evaluate TRANSPARENT's false positive rate and is located at the `accuracy/fdr/` directory.

B. Artifact Installation & Configuration

To install all the software dependencies for the evaluation, follow these steps:

1) Install the Nix package manager (see https://nixos.org/download/)
2) Install and activate Git LFS (see https://github.com/git-lfs/git-lfs)
3) Download our artifacts (through the DOI or by running `$ git clone –recursive https://github.com/diwangs/transparent-ae`)
4) Change working directory to `transparent-ae`
5) Run the install script (`$ ./install.sh`)

To check whether the software are installed correctly, you could run the unit tests of TRANSPARENT by doing the following:

1) Change working directory to `transparent`
2) Run the test script (`$ ./test.sh`)

C. Major Claims

- (C1): TRANSPARENT is able to reveal SPA-specific sinks, including novel ones that are not included in the state-of-the-art tool, vanilla CodeQL. This is proven by the experiment (E1) whose results are reported in Table V in the paper.
- (C2): TRANSPARENT is able to reveal bugs in real-world SPA and achieve better accuracy than vanilla CodeQL. This is proven by the experiment (E2) whose results are reported in Table IV in the paper.

D. Evaluation

1) Experiment (E1): [Automated Framework Abstraction] [30 human-minutes + 2 compute-hours]: Run TRANSPARENT on Vue, React, and Angular to obtain its SPA-specific sink

[Preparation] Change working directory to `accuracy`.

[Execution] Run `$ ./main.sh`

[Results] After the script is finished running, Table V should appear in `stdout`. Additionally, the synthesized CodeQL query library used for the next experiment should be outputed at `../qlpack/transparentsinks`

2) Experiment (E2): [Accuracy Evaluation] [30 human-minutes + 4 compute-hours]: Run the synthesized query obtained from the previous experiment and apply them to both known vulnerabilities dataset and positive label GitHub dataset.

[Preparation] Change working directory to `accuracy`.

[Execution] Run `$ ./main.sh`

[Results] After the script is finished running, Table IV should appear in `stdout`. Analysis of each repository would be generated on `accuracy/fnr/build` and `accuracy/fdr/build`. Each json file consists of alerts in a given repository generated by either TRANSPARENT or vanilla CodeQL (signified with `_t` and `_b` suffix respectively.)