ReactAppScan: Mining React Application Vulnerabilities via Component Graph

Zhiyong Guo
Johns Hopkins University
Baltimore, MD, USA
zguo55@jh.edu

Mingqing Kang
Johns Hopkins University
Baltimore, MD, USA
mkang31@jhu.edu

V.N. Venkatakrishnan
University of Illinois Chicago
Chicago, IL, USA
venkat@uic.edu

Rigel Gjomemo
University of Illinois Chicago
Chicago, IL, USA
rgjome1@uic.edu

Yinzhi Cao
Johns Hopkins University
Baltimore, MD, USA
yinzhi.cao@jhu.edu

ABSTRACT

React, a single-page application framework, has recently become popular among web developers due to its flexible and convenient management of web application states via a syntax extension to JavaScript, called JSX (JavaScript and XML). Despite its abundant functionalities, the security of React, especially vulnerability detection, still lags: many existing vulnerability detection works do not support JSX let alone React Data Flow introduced by React components. The only exception is CodeQL, which supports JSX syntax. However, CodeQL cannot properly track React Data Flow across different components for detecting vulnerabilities.

In this paper, we design a novel framework, called ReactAppScan, which constructs a Component Graph (CoG) for tracking React Data Flow and detecting vulnerabilities following both JavaScript and React data flows. Specifically, ReactAppScan relies on abstract interpretation to build such a component graph via tracking component lifecycles and then detects vulnerabilities via finding paths between sources and sinks. Our evaluation shows that ReactAppScan detects 61 zero-day vulnerabilities in real-world React applications. We have responsibly reported all the vulnerabilities and so far six vulnerabilities have been fixed and two have been acknowledged.

CCS CONCEPTS

• **Security and privacy → Web application security**.

KEYWORDS

Single-page Application; Vulnerability Detection; Component Graph

ACM Reference Format:
Zhiyong Guo, Mingqing Kang, V.N. Venkatakrishnan, Rigel Gjomemo, and Yinzhi Cao. 2024. ReactAppScan: Mining React Application Vulnerabilities via Component Graph. In *Proceedings of the 2024 ACM SIGSAC Conference on Computer and Communications Security (CCS '24), October 14–18, 2024, Salt Lake City, UT, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3658644.3670331

CCS '24, October 14–18, 2024, Salt Lake City, UT, USA

ACM ISBN 979-8-4007-0636-3/24/10
https://doi.org/10.1145/3658644.3670331

1 INTRODUCTION

Single-page applications (SPAs) [53]—which allow websites to interact with users via a single HTML page—have recently become very popular in web application designs. Famous SPAs include many widely-used websites such as Facebook, Gmail, Twitter, and GitHub. One notable framework for building SPAs is called React (or called React.js or ReactJS) [25], which is used by over 13 million live websites [40] and is being voted as the second most popular web frameworks [5] only falling behind Node.js (which often serves as the foundation of React and is not an SPA) on Stack Overflow. Specifically, React uses a syntax extension to JavaScript, called JSX (JavaScript and XML), which embeds HTML snippets as part of JavaScript and models them as components [34], thus reducing web developers' efforts in maintaining and synchronizing state.

While React has revolutionized web application design, React applications—just like traditional web applications—may still be vulnerable to classic vulnerabilities such as Cross-site Scripting (XSS) [67, 72, 83]. However, many state-of-the-art works on web application vulnerability detection, such as FAST [59] and ODGen [69], cannot detect React application vulnerabilities. On one hand, they do not natively support the analysis of JSX code. Fundamentally, such support is *challenging* because of so-called React Data Flow [19], which passes data between different React components, e.g., between parent and child or between siblings, via Props [24] and State [31] indirectly. On the other hand, their analysis cannot scale to JavaScript code that is transpiled from even simple JSX code due to state explosion according to our experiment.

CodeQL is a commercial tool that supports JSX syntax and that can detect some React application vulnerabilities [10]. However, CodeQL does not properly support the aforementioned React Data Flow, making it unable to detect many real-world vulnerabilities. The support of React Data Flows is challenging because CodeQL's representations of objects are coarse-grained, lacking the understanding of props and state in different components. We reported the issue together with test cases to CodeQL developers. They consider the problem challenging [33], because a fix may "blow up [their analysis] in complexity/runtime" and lead to "possible [large] false positives". Eventually, CodeQL made an update, which is the version used in our evaluation, but it still performs very poorly in detecting real-world vulnerabilities with large false negatives.

In this paper, we design a framework, called ReactAppScan, to mine React application vulnerabilities via a so-called Component

```
1 function Comp (props) {
2   const [html, setHtml] = useState('');
3   useEffect(() => {
4     fetch('https://api.example.com/data')
5       .then(res => res.json())
6       .then(data => setHtml(data));
7   }, []);
8   return <div dangerouslySetInnerHTML={{ __html: html
         }} />; };
```

Figure 1: A simple code snippet that illustrates a React component

Graph (CoG). Our *key* idea is to represent React components together with props and state in a graph so that one object instance—no matter as props or state of different components—has only one node representation but multiple edges from different props or state in the graph. Then, ReactAppScan queries the graph for paths between sources (e.g., HTTP requests) and vulnerability-specific sinks (e.g., `dangerouslySetInnerHTML`) to detect vulnerabilities.

Specifically, ReactAppScan builds CoGs via abstract interpretation following React component lifecycles. That is, first, ReactAppScan constructs an initial CoG via parsing the `return` statements of JSX and abstractly interprets the render function of each component. Next, ReactAppScan monitors the state and props changes of each component to abstractly interpret the render or lifecycle methods/hooks using a queue-like structure, should changes be observed, mimicking the updating phase. Lastly, ReactAppScan also simulates the unmounting stage of React components.

Our implementation of ReactAppScan is open-source [27] and we run ReactAppScan upon popular React applications on both GitHub and NPM. Our evaluation results in 61 zero-day vulnerabilities. We have responsibly reported all the findings to their developers: So far, six vulnerabilities have been fixed and two additional have been acknowledged. We also compared our approach with the improved version of CodeQL on two datasets, including one with real-world GitHub and NPM applications and another with known CVE vulnerabilities. Our evaluation shows that ReactAppScan has fewer false positives and negatives than CodeQL.

We make the following contributions in the paper:

- We designe the *first* abstract interpretation framework of JSX, called ReactAppScan, to model React Data Flow using a component graph and detect React application vulnerabilities.
- ReactAppScan models and tracks client-server communication to detect vulnerabilities that span both sides, e.g., those originating from a client adversary, traversing through a victim server, and ending in a client victim.
- Our evaluation shows that ReactAppScan detects zero-day vulnerabilities of real-world React applications from GitHub and NPM and outperforms the state-of-the-art vulnerability detection tool, namely CodeQL.

2 BACKGROUND

In this section, we give a background of React and React-specific terminologies using a simple code snippet in Figure 1 for readers unfamiliar with React.

React Components. A React component describes the User Interface (UI) of a web application and its purpose is to return HTML to a web page. There are two types of React components: (i) function component and (ii) class component. First, a function component, starting with an uppercase first letter, returns a React element, i.e., a JavaScript object describing a DOM node and its properties. Figure 1 shows a function component with the definition at Line 1, and the return statement is at Line 8. Second, a class component, extending the Component class from React library, has a `render` method that returns a React element. React components form a tree-like structure based on the return statement just like a Document Object Model (DOM) tree.

There are two important objects of each React component and we describe them below:

- *Props.* Props [24] describe any inputs that are passed to a React component, which usually comes from a parent component. The first argument of a function component is the props, e.g., at Line 1 of Figure 1; the constructor of a class component receives a props argument and passes it to the parent constructor using the super keyword. A constructor of a class component can be omitted if there are no other purposes.
- *State.* State [31] in React is mutable data that changes when a user interacts with the web application; when state changes, React components are re-rendered to update their UIs. The original design of React is to use React class components to hold state, such as "this.state"; since React 16.8, a function component can use "Hooks", such as "useState" (Line 2 of Figure 1), to hold state as well.

React Data Flow. React Data Flow is unidirectional, i.e., the data goes down from parent to child components via props; instead, user-triggered actions and the follow-up updates go up, creating a circular system. This follows React's philosophy: the user triggers actions that modify the state of a React application, which then alters the UI. For example, the "html" prop at Line 2 of Figure 1 shows a data flow that passes the "html" data from a parent component, i.e., "Comp", to a child, i.e., a HTML div tag, whose attribute 'dangerouslySetInnerHTML " is also a Cross-site Scripting (XSS) sink.

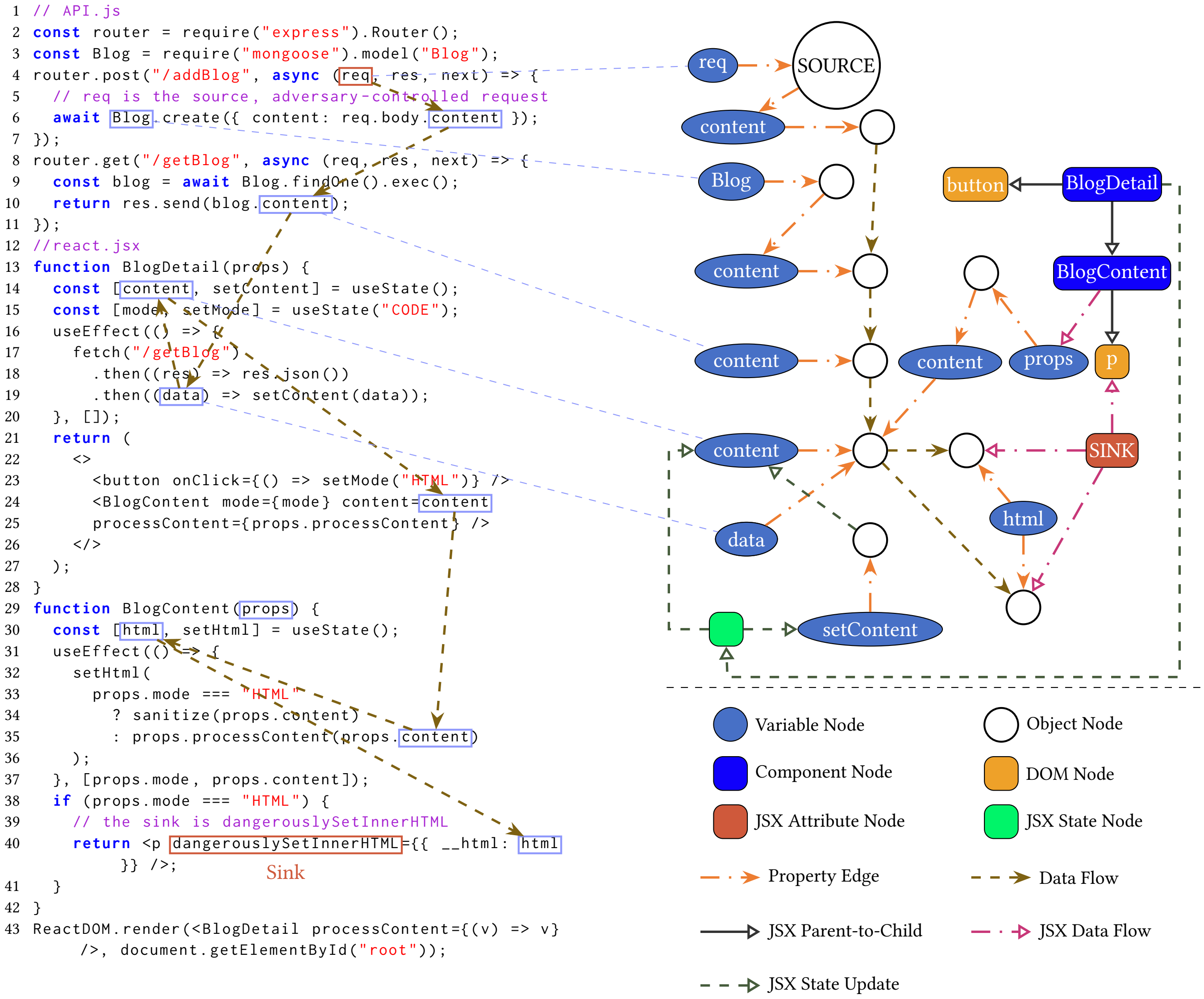
Each React component has a lifecycle, i.e., starting from mounting, to updating and then to unmounting. A function component uses "useEffect" (Line 3 of Figure 1), i.e. React hooks, to hold state and monitor state changes in a lifecycle. A class component has many lifecycle-related methods, e.g., `componentWillMount` (which is invoked immediately before the component is inserted into the DOM) and `componentDidMount` (which is invoked immediately after the component is inserted into the DOM).

3 OVERVIEW

In this section, we start from a motivating example in Section 3.1 and describe our threat model in Section 3.2.

3.1 A Motivating Example

Figure 2 illustrates a React application built with MongoDB [21], Express.js [14], React, and Node.js, i.e., the so-called MERN technique. The application—motivated by a real-world XSS vulnerability (CVE-2023-22462 [6]) and adapted for easy description—is a blogger, which allows users to add blogs via `addBlog` (Line 4) and read blogs



```
1  // API.js
2  const router = require("express").Router();
3  const Blog = require("mongoose").model("Blog");
4  router.post("/addBlog", async (req, res, next) => {
5    // req is the source, adversary-controlled request
6    await Blog.create({ content: req.body.content });
7  });
8  router.get("/getBlog", async (req, res, next) => {
9    const blog = await Blog.findOne().exec();
10   return res.send(blog.content);
11 });
12 //react.jsx
13 function BlogDetail(props) {
14   const [content, setContent] = useState();
15   const [mode, setMode] = useState("CODE");
16   useEffect(() => {
17     fetch("/getBlog")
18       .then((res) => res.json())
19       .then((data) => setContent(data));
20   }, []);
21   return (
22     <>
23       <button onClick={() => setMode("HTML")} />
24       <BlogContent mode={mode} content={content}
25       processContent={props.processContent} />
26     </>
27   );
28 }
29 function BlogContent(props) {
30   const [html, setHtml] = useState();
31   useEffect(() => {
32     setHtml(
33       props.mode === "HTML"
34         ? sanitize(props.content)
35         : props.processContent(props.content)
36     );
37   }, [props.mode, props.content]);
38   if (props.mode === "HTML") {
39     // the sink is dangerouslySetInnerHTML
40     return <p dangerouslySetInnerHTML={{ __html: html
          }} />;
41   }
42 }
43 ReactDOM.render(<BlogDetail processContent={(v) => v}
       />, document.getElementById("root"));
```

Figure 2: A motivating example with a Cross-site Scripting (XSS) vulnerability (Line 40), which is simplified from CVE-2023-2246 [6] for the description purpose.

via `getBlog` (Line 8). Then, `react.jsx` (Lines 12–43) of the application provides a user interface with different React components, such as *BlogDetail* (Line 13) and *BlogContent* (Line 29).

A successful exploit of the XSS vulnerability starts from a malicious request to the `addBlog` API from an adversary until the `dangerouslySetInnerHTML` sink (Line 40). The adversary-controlled data is stored in MongoDB (Line 6) and read by a benign user request to the `getBlog` API. Then, the data is stored as a state of the `BlogDetail` component (Line 13) as `content` (Line 14) and then passed to the `BlogContent` component (Line 29) as a `props` and finally to the sink (Line 40).

Research Challenges. There are three main research challenges in detecting this XSS vulnerability.

- *React Data Flow.* There are two React Data Flows in this application making the vulnerability challenging to detect. First, let us start from the data flow related to `content` at Line 14. The flow starts from setting a state of the `BlogDetail` component (Line 19) and then goes into a prop of the `BlogContent` (Line 24) and then a prop of the `p` tag (Line 40). This is a challenging data flow because the flow depends on the `useEffect` hook (Line 31) and another state (i.e., `mode` at Line 15) in the *BlogDetail* component. In other words, the application is only vulnerable after the hook (Line 31) is invoked and `mode` is set as "HTML". Second, we describe the data flow related to `processContent` at Line 43. This `processContent` function is defined as a prop of the `BlogDetail` component (Line 43), passed to the `BlogContent` component as another prop (Line 25), and then eventually invoked at Line 35. None of the existing works [10, 59, 69] can track both data flows, let alone detect the XSS vulnerability, due to the cross-component nature of both flows.

- *Client-server Data Dependency.* The data dependency between `blog.content` at Line 10 in "API.js" and `res/data` at Line 18/19 in "react.jsx" is due to client-server communication via the `fetch` at Line 17. This is important because a server response may not be controllable by an adversary (e.g., it could be a constant value) and such a data dependency links the server response to another client's request, i.e., `req` at Line 4, which is controllable by an adversary. Existing works [10, 59, 69] do not track such cross-side data dependencies, which leads to false positives because some server responses are not controlled by an adversary.
- *Database-related Data Dependency.* The data dependency between `req.body.content` (Line 6) and `blog.content` (Line 10) is caused by MongoDB, a NoSQL database. This is a challenging task because one needs to map the store operation using the `content` keyword (Line 6) with the access operation using the same keyword. Again, none of the existing works [10, 59, 69] models such a database-related data dependency.

Our Key Idea: Component Graph (CoG). We describe our idea in detecting the XSS vulnerability in Figure 2. In a nutshell, our objective is to find data flows from user input (i.e., the req object at Line 4) to sensitive sinks (i.e., dangerouslySetInnerHTML at Line 39) in detecting this XSS vulnerability. However, to be able to find these data flows successfully, we need to solve the aforementioned three types of challenging data dependencies.

Now, we describe how ReactAppScan solves these three research challenges. First, let us start with the challenge of modeling React Data Flows. ReactAppScan models React components as a CoG as shown on the right part of Figure 2. All components, e.g., `BlogDetail` and `BlogContent`, are modeled as nodes following their parent-child relations and then the states and props of components are also represented as nodes under the component nodes. Note that objects with aliases are represented as the same node: For example, ReactAppScan only maintains one single node for the `content` state of the `BlogDetail` component and the `content` prop of the `BlogContent` component. This also follows React logic because once the state of `BlogDetail` changes, the prop of `BlogContent` changes as well automatically. Second, we describe how we solve the challenges of the client-server and database-related data dependencies. ReactAppScan records the key used in such data dependencies, e.g., the `content` key used for the database at Line 6 and the `/getBlog` key for the server router at Line 8 and the client fetch at Line 17. Then, ReactAppScan links the corresponding data in a database or a network request/response based on the common key and annotates them in the CoG.

ReactAppScan builds this CoG with these challenging data dependencies via abstract interpretation with the abstract domain as the graph. The building starts with the static structure of React components in JSX and then models the updating procedure just like what React does. For example, if a prop to a component has changed, ReactAppScan will abstractly interpret the function component definition or the render method of a class component.

The proposed CoG is complementary to and can be combined with existing program analysis data structures, such as Object Dependence Graph (ODG) [69], Code Property Graph (CPG) [89], or Program Dependency Graph (PDG) [52], for vulnerability detection. That is, CoG models data flows between React components that are not modeled by existing structures, and such modeled data flows can be connected with the rest data flows in existing structures. Take ODG for example. Figure 2 shows that the data flow starts from `req.content`, i.e., an ODG node, passes through a few ODG nodes, reaches a `state` node of `BlogDetail`, and then ends up with an attribute node of the `p` tag, i.e., the 'dangerouslySetInnerHTML' attribute.

3.2 Threat Model

In this subsection, we describe our threat model. The victim in our threat model is a vulnerable React application, which can contain a vulnerability on either the client- or the server-side. In-scope vulnerabilities are XSS, arbitrary file upload, and improper authorization. Then, the adversary in our threat model could be one of the following:

- A malicious client. The adversary attacks the victim server of the vulnerable React application by sending a malicious request, which could result in exploiting the server or the client, for instance, using an XSS payload. Our motivating example in Figure 2 is such a case, where the adversary sends a malicious request as the source.
- A crafted victim URL. The adversary tricks a victim client into visiting a URL belonging to the victim server with a crafted input as part of the URL parameter. Such a parameter may trigger a vulnerability on the client side, e.g., a DOM-based XSS with URL parameters as the source.
- A malicious website. The victim may accidentally visit a malicious application, e.g., by visiting a malicious URL, causing the adversary-controlled website to be loaded in the same browser as the vulnerable React website, e.g., in different tabs. Then, the malicious website sends a message (e.g., via `postMessage`) to attack the React website, which could lead to improper authorization and trigger another vulnerability, e.g., XSS.

We also classify existing vulnerabilities into two categories following prior works [59, 69], which are (i) application-level and (ii) package-level. The former allows an end-to-end attack from an adversary to a vulnerable sink, e.g., from either a malicious client request or a malicious message to the sink. The latter exposes an external API without proper sanitization, which makes another application using the package potentially vulnerable. Such vulnerabilities are very common and well-documented in the CVE database [1, 2, 6, 7].

4 DESIGN

In this section, we describe the system architecture of ReactAppScan and then present the detailed three phases of ReactAppScan.

4.1 System Architecture

Figure 3 shows the overall architecture of ReactAppScan, which takes the source code of a React package or application as input and outputs detected vulnerabilities. The high-level idea is that ReactAppScan follows the rendering process of native React on an application to abstractly interpret its code and to build a CoG, which can be queried for vulnerability detection.

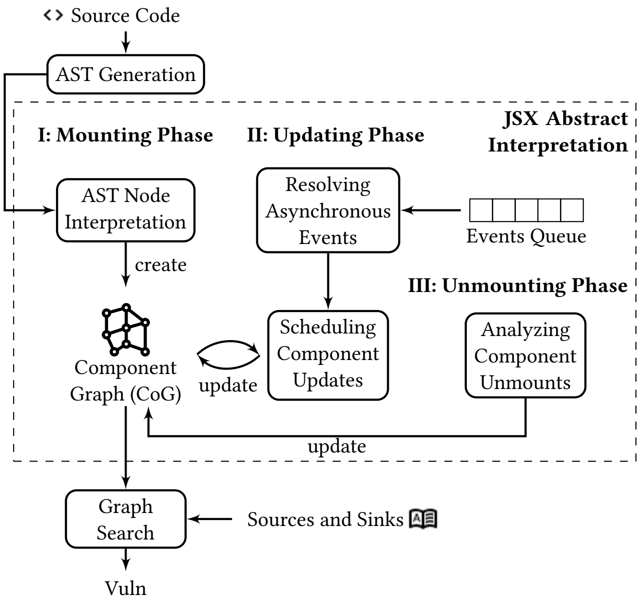


Figure 3: System Architecture

Following the lifecycles of React components, naturally, there are three phases for the detection: (i) mounting, (ii) updating, and (iii) unmounting. First, in the mounting phase, ReactAppScan builds an initial CoG based on the static JSX file. Specifically, ReactAppScan starts from the entry points of the Abstract Syntax Tree (AST) and abstractly interprets each AST node with modeled React.js APIs and client-side APIs to generate this CoG. ReactAppScan also queues asynchronous callbacks for preparation of the next phase. Second, in the updating phase, ReactAppScan processes asynchronous callbacks and hooks/lifecycle methods, and then updates the CoG based on prop and state updates by abstractly interpreting the render method of the component that needs to be updated. Third, in the unmounting phase, ReactAppScan looks up clean-up functions or unmount methods to simulate the unmounting process. In the end, after three phases, ReactAppScan queries the graph for an unsanitized path between an adversary-controlled source and a vulnerability-specific sink to detect vulnerabilities.

Now consider the simple example in Figure 1. ReactAppScan first constructs an initial CoG during the mounting phase, in which the state node "html" (Line 2) points to an empty string. ReactAppScan also queues the asynchronous callback function, notably the "useEffect" function at Line 3, for the second phase. Second, in the updating phase, ReactAppScan abstractly analyzes the queued asynchronous callback, i.e., adding a link from state node "html" to the network response. Lastly, in the unmounting phase, ReactAppScan abstractly interprets cleanup function, which does not exist in our simple example. After the CoG is built, ReactAppScan queries the graph to find an unsanitized path between the source (i.e., "res" at Line 5) and the sink (i.e., "dangerouslySetInnerHTML" at Line 8).

We describe these steps in more details next.

Table 1: Notations (e.g., nodes, edges, and procedures) of Component Graph

Notations	Descriptions
N	A set of component graph nodes
$el \in N_{el} = N_c \cup N_d$	JSX element (DOM or component node)
$c \in N_c$	A JSX Component Node
$d \in N_d$	A DOM element node
$\mathtt{state} \in N_{\mathtt{state}}$	The state node of a JSX component
$\mathtt{props} \in N_{\mathtt{props}}$	The props node of a JSX component
$\mathtt{attr} \in N_{\mathtt{attr}}$	A JSX Attribute Node of a JSX Element
$a \in N_{\mathsf{AST}}$	An AST Node
$v \in N_{\mathtt{var}}$	A variable Node
$o \in N_{\mathtt{obj}}$	A JSX Object Node
E	A set of component graph edges
$el \rightarrow a$	The AST node (a) defines the element el
$c \rightarrow \mathtt{state}$	The edge between a component and its state
$c \rightarrow \mathtt{props}$	The edge between a component and its props
$\mathtt{state} \rightarrow < v, v_f >$	A state variable v and its setState function v_f of a state belonging to a certain component.
$\mathtt{props} \rightarrow v$	A prop variable v of a props node belonging to a certain component.
$el \rightarrow \mathtt{attr}$	An attribute node belonging to a JSX element
$el \rightarrow el$	Parent-child JSX element relation.
$v/\mathtt{attr} \rightarrow o$	The object of a variable or a JSX attribute
$o \rightarrow o$	JSX data dependency
$\mathtt{o} \rightarrow v$	The attribute of an object
JSX Procedures (N)	*All the JSX related operations*
$\mathtt{Child}^{\mathtt{EdgeType}}_{\mathtt{parentNode}}$	Get the child node of `parentNode` with `EdgeType`
$\mathtt{AddXXX}^{a}_{\mathtt{name}}$	Add a JSX component/DOM/element/attribute node `name` and AST node a (i.e., `XXX` = `Comp`, `DOM`, `El`, `Attr`).
$\mathtt{AddNode}^{\mathtt{NodeType}}_{a}$	Add a node from a with `NodeType`.
$\mathtt{AddEdge}^{\mathtt{EdgeType}}_{\mathtt{src} \rightarrow \mathtt{dst}}$	Add an edge from `src` to `dst` with `EdgeType`.
$\mathtt{AddProperty}^{o_1 \rightarrow o_2}_{\mathtt{name}}$	Add object o_2 as a property of object $o1$ with the `name` of property.
$\mathtt{Copy}(o_1, o_2)$	Copy object o_1 to $o2$. For each property in o_2, add an object as a property of o_1 with the same name. Furthermore, data flow is added from o_1 to o_2 for these properties.
$\mathtt{HasCommonProperty}(o_1, o_2)$	Check if object o_1 and object o_2 have any common property names, if o_2 has any properties.
$\mathtt{LkupName}(a)$	Get the name of a JSX Element with its AST a
$\mathtt{LkupAttr}(a)$	Look up a JSX Attribute Node by the AST node a.
$\mathtt{LkupXXX}(c)$	Look up the state/state object/props object/state variable/prop variable node of a component c (i.e., `XXX` = `State`, `StateObjs`, `PropsObjs`, `StateVar`, `PropsVar`.
`LkupMountingFunc(c)`	Look up the mounting lifecycle methods of a component c, which include the function component definition, `constructor`, `getDerivedStateFromProps`, `render`, and `componentDidMount`.
`LkupUpdatingFunc(c)`	Look up the updating lifecycle methods of a component c, which include the function component definition, `getDerivedStateFromProps`, `shouldComponentUpdate`, `componentDidUpdate`, `getSnapshotBeforeUpdate`, and `render`.
`LkupCleanupFunc(c)`	Look up the cleanup lifecycle methods of a component c, which include the cleanup function definition of `useEffect` and `componentWillUnmount`.
`Compare(c)`	Compare whether the props object or the state object of a component changes.

4.2 Phase I: Mounting

We first describe the definition of a component graph and then the abstract interpretation process to build such a component graph.

4.2.1 Definitions and Notations. We define a Component Graph as a graph with JSX-related objects and variables (e.g., JSX elements, JSX states, and JSX props) as nodes (N) and their relations as edges (E). Table 1 describes the nodes and edges of a CoG. The core part of a CoG is a tree-like structure consisting of different JSX elements, i.e., either a JSX component or a DOM element, with their attributes, which is similar to a DOM tree but with JSX components as well. Each JSX component node has a `state` node representing its internal states and a `props` node representing attributes passed from its parent component. Then, variable nodes are under `state` or `props` nodes and may point to different objects or to the same object (e.g., the `content` prop under `BlogContent` and the `content` state under `BlogDetail` pointing to the same object in Figure 2).

As discussed, one of the main advantages of a CoG is that it can be combined with existing established program analysis data structures, such as Object Dependence Graph (ODG) [69], Code Property Graph (CPG) [89], or Program Dependency Graph (PDG) [52]. The combination with ODG, PDG, or CPG follows the data flow: In our example in Figure 2, ODG, PDG, or CPG handles the previous, classic data flow, and our CoG models the data flow related to React to the final 'dangerouslySetInnerHTML' sink, i.e., a JSX attribute.

4.2.2 Operational Semantics. We now provide the overview of selective operational semantics across the mounting, updating, and unmounting phases. The complete operational semantics is in Figure 8 of Appendix A. The abstract domain state is denoted as a tuple $p = (N, E, el, q, S)$, where N represents all nodes, E represents all edges, el is the current JSX element being interpreted, and q is the queue for scheduling rendering and lifecycle methods. S is a global state that records the snapshot, i.e., the props and state of a component. It also handles registering and discovering network response callbacks. Note that all AST node definitions in the operational semantics follow the JSX specification [3]. There are four different categories of operational semantics in generating CoG for JSX and we describe them below.

- Analyzing JSX elements to generate a Tree-like Structure. ReactAppScan abstractly interprets `JSXElement` to add JSX elements into the CoG. Adhering to the naming rule of JSX components [3], if the name of a `JSXElement` begins with a capitalized letter, ReactAppScan adds a JSX Component node c to the graph. Otherwise, if the name starts with a lowercase letter, ReactAppScan adds a DOM node d. Next, the interpretation of `JSXChildren` establishes parent-child relationships between JSX elements. Specifically, if $\texttt{JSXElement}_i$ appears in the `JSXChildren` of another $\texttt{JSXElement}_j$, ReactAppScan adds a parent-child relation $\texttt{JSXElement}_j \longrightarrow \texttt{JSXElement}_i$.
- Analyzing JSX attributes and props to model data flows between JSX Elements. ReactAppScan models the data flow between JSX elements through JSX attributes and props. A JSX attribute is comprised of a `JSXAttributeName` and a `JSXAttributeValue`. ReactAppScan abstractly interprets the AST children of name and value separately, yielding attribute name and object nodes for the value. Then a JSX attribute node `attr` with the attribute name is added, with an edge pointing to el. Additionally, ReactAppScan adds JSX Data dependency edges to link the JSX attribute node to object nodes. We then describe a specific JSX attribute, `ref`, which provides access to the DOM. `useRef` returns an object node with a property named `current`. The `ref` is linked with a DOM node when it is passed to the JSX attribute `ref` of a DOM node. Consequently, any write operation to `current` is seen as a write to the DOM, which leads to XSS. Next, ReactAppScan also models objects passed into a component via `props`. Each JSX component has a reference to its `props`. When rendering, ReactAppScan either creates `props` on first render or updates the `props`. ReactAppScan adds `JSXAttributeValue` objects as properties to `props`, using the JSX attribute names as keys.
- Analyzing JSX states to model state-related data flows. ReactAppScan models data flow within a JSX component using state nodes. Each JSX component maintains a reference to a state node, denoted as $state$. This node links state variables v and corresponding `setState` functions v_f. When v_f is invoked, ReactAppScan resolves the arguments passed to v_f and updates v to point to the argument's objects.
- Modeling JSX component rendering. ReactAppScan first looks up the definition function for function components, or the mounting functions for class components. It then invokes these functions with the necessary arguments, specifically, the props and state objects as required.

4.3 Phase II: Updating

After ReactAppScan builds an initial CoG, the next phase, called updating, is to update the CoG based on asynchronous events and JSX hooks/lifecycle methods as described in the operational semantics for this phase. The full list is in Figure 8 of Appendix A.

4.3.1 Graph Updates for Asynchronous Events. ReactAppScan maintains a queue structure that stores asynchronous callbacks, such as a DOM event listener, during abstract interpretation in the first phase (mounting). Once the first phase is done, ReactAppScan fetches all the callbacks from the queue to analyze them sequentially. Detailed operational semantics are shown in the "Async Events" part of Figure **??**. There are two special cases for such callbacks:

- Network response callbacks. ReactAppScan introduces a service registry to maintain a relationship between each network request call (e.g., AJAX) and its corresponding target function. Such an analysis of network responses follows a three-step process: First, ReactAppScan adds the registration of service functions to the service registry. Specifically, ReactAppScan abstractly interprets the API route's AST nodes with the modeled Node.js APIs and framework APIs and records the API key and corresponding function definition in the process. Second, ReactAppScan discovers the service functions when abstractly interpreting the React.js AST nodes. During this stage, when processing an AJAX or fetch call, ReactAppScan matches the URL in the service registry to find the target function recorded and call it. ReactAppScan precisely matches static paths in routes, and also aligns variables parts with placeholders in dynamic routes. Third, after invoking the function, the points-to information between the variable in the React.js code and the object returned by the API is modeled. Therefore, ReactAppScan establishes a server-client data dependency.
- Database-related callbacks. ReactAppScan handles database-related callbacks leveraging the database model semantics, supporting Create, Read, Update, and Delete (CRUD) operations. Each database model, such as the Blog model in Figure 2 (Line 2), is represented as an object node in the CoG. The create operation, such as 'Blog.create' at Line 6, along with update operation, establish object-level data flow from input to the model's properties. Subsequently, read operations, for instance, 'Blog.findOne' at Line 9, create data flow from the model's properties to the corresponding properties of the returned object. Note that some

data operations may involve query filters, which are JavaScript objects that define fields with keys and set conditions with values, as utilized in Object Data Modeling (ODM) libraries like Mongoose [22]. If any key is specified in the query, ReactAppScan constructs a regular expression by joining model keys with 'or' operators between them. This regular expression is then used to test against the query keys to check for the presence of any common keys between them. If found, ReactAppScan creates data flow.

4.3.2 Graph Updates for JSX Component Updates. ReactAppScan updates CoG based on updates of JSX components, e.g., new props and state updates. Detailed operational semantics are shown in Figure 8 of Appendix A. We divide this process into two parts: (i) update condition determination, and (ii) CoG updates. First, ReactAppScan determines which components require updating based on three different conditions:

- New Props passed to a component. ReactAppScan checks this case by comparing whether the props object of a component changes based on snapshots. Specifically, ReactAppScan takes snapshots of all the props belonging to JSX component before and after each update. The initial "before" snapshot is the one after Phase I (Mounting) but before analyzing the asynchronous callbacks and the initial "after" snapshot is the one after analyzing the asynchronous callbacks. ReactAppScan compares two snapshots by examining their properties via property edges. If there is a change detected in any properties of the props objects, including the addition of a new property and a property pointing to a new object, ReactAppScan concludes that the component needs updates.
- `setState` method call. When `setState` is called inside a component, which can be either the `setState` function in function components or the `this.setState` function in class components. Upon the invocation of `setState`, ReactAppScan first updates state node by pointing the state variable to resolved objects of `setState` arguments. Then it finds the associated component via the JSX state update edge and marks it for updates.
- `forceUpdate` method call. When the `forceUpdate` API is invoked, it serves as a method to forcibly update a component in React.js. Upon calling `forceUpdate`, ReactAppScan finds the associated component's updating functions except for the method `shouldComponentUpdate` and marks the component for a forced update.

Second, ReactAppScan finds all the updating function definitions via `LkupUpdatingFunc`. For function components, ReactAppScan finds the function definition and the effect-related methods. For class components, ReactAppScan finds the lifecycle methods by looking up the function definitions with specific lifecycle method names, adhering to the sequence prescribed by React lifecycle.

Third, ReactAppScan abstractly analyzes these updating functions. For function components, the component definition is executed with the current props and state objects. During analysis of effect-related functions, such as `useEffect`, ReactAppScan enqueues the callback function. For class components, the analysis is based on argument types. ReactAppScan analyzes `Constructor`, `getDerivedStateFromProps`, `shouldComponentUpdate`, as well as `render` with current props and state objects; then, ReactAppScan analyzes `getSnapshotBeforeUpdate` and `componentDidUpdate` with the previous props and state objects, which are stored as snapshots in the global state S. Such steps will be iterated until convergence (i.e., ReactAppScan calls the lifecycle methods and repeats the process from the first step until no more changes are observed for the CoG) or exceeding a maximum number of iterations.

4.4 Phase III: Unmounting

After the updating phase, the CoG is updated based on unmounting of JSX components. The operational semantics of this process are also shown in Figure ??. ReactAppScan looks up cleanup functions, including cleanup effects for function components, specifically the returned function of the first argument of `useEffect`, and `componentWillUnmount` for class components. Following this, ReactAppScan abstractly analyzes these functions to update the CoG.

5 IMPLEMENTATION

Our implementation, comprising 4,689 lines of new code excluding any third-party code (e.g., those mentioned below), is open-source and can be accessed at an anonymous repository [27]. Our Abstract Syntax Tree (AST) parser of JSX is based on an open-source tool, called Espree [13]. Next, our abstract interpretation of JavaScript is based on open-source repositories of both ODGen [4] and FAST [59]: Specifically, we reuse the representation and generation of ODG and the modeling of built-in functions from these sources to model JavaScript features, notably dynamic features such as prototype chain, reflection, and dynamic property lookups. In addition, ReactAppScan abstractly interprets all branches in parallel as does ODGen. We included the improvement in FAST over ODGen (e.g., Promise) into ODGen, but did not use its two-phased abstract interpretation because JSX sinks are JSX attributes rather than JavaScript function calls. Note that none of ODGen or FAST code is included in our Line of Code count. Currently, our implementation supports all React features in its version 16, the most prevalent as per W3Techs reports [84] as well as popular features in React versions 17 and 18 (e.g., those related to React data flows).

Furthermore, our implementation adopts the graph query function of ODGen, i.e., a depth-first search (DFS) function to find paths from sources to sinks. There are two improvements for vulnerability detection of React vulnerabilities. First, ReactAppScan adopts a customized list of sources and sinks as shown in Table 2. Note that ReactAppScan does not include the setting of innerHTML for the `<script />` tag as a sink. This is because, according to HTML standards, script elements inserted using `innerHTML` should not execute [15]. We apply the same rule to the `<style />` tag. Note that AJAX requests are categorized as sinks when an attacker can manipulate the request URL, enabling the execution of a privileged AJAX call, as seen in CVE-2023-5654 [8]. Second, ReactAppScan models popular sanitization libraries such as dompurify [12], markdown-it [20], and sanitize-html [30] during graph query for vulnerability detection. That is, if a sanitization function is present between the source and sink, ReactAppScan considers this path as not vulnerable.

Table 2: A List of Sources and Sinks

Type	APIs
Application-level Sources	
Network Request	HTTP(S) requests server packages, e.g., Express.js
URL	window.location useSearchParams() (react-router-dom)
Message	`message` event
Package-level Sources	
Exported APIs	function arguments of `module.exports (Node.js)` and `export (ES2015)`
Sinks	
DOM Write	dangerouslySetInnerHTML Setting innerHTML of a DOM Element document.write
Location Functions	location.replace location.assign Setting location.href window.open
AJAX Requests	fetch axios
DOM Attribute Sinks	<a href /> <form action /> <iframe src /> <area href /> <button formaction /> <input formaction /> <frame src />

6 EVALUATION

In this section, we evaluate ReactAppScan using the following research questions:

- RQ1: How many zero-day vulnerabilities can ReactAppScan detect in real-world React applications (but state-of-the-art approaches cannot)?
- RQ2: What are the false positives and negatives of ReactAppScan when compared with state-of-the-art approaches (e.g., CodeQL)?
- RQ3: What are the performance overhead and code coverage of ReactAppScan in analyzing React applications?

6.1 Experimental Setup

In this subsection, we describe our experimental setup including the datasets and the experimental environment used in the evaluation.

6.1.1 Datasets. We prepare two datasets for evaluating false positives and negatives separately.

- Large-scale unlabelled dataset consisting of real-world React applications (called Large-scale Dataset). There are two sources of this dataset: (i) GitHub and (ii) NPM. First, we use the GitHub API to crawl 6,382 repositories built using React technologies in November 2023. Specifically, we search repositories with "react" as a topic and having more than 10 stars. We then keep those repositories that have React.js libraries as dependencies. Second, we also crawled NPM to find 4,122 React packages with weekly downloads that were larger than 1,000 in November 2023. Specifically, we identify a React package based on the presence of a package.json file that specifies "react" within any of the three dependency fields: dependencies, devDependencies, or peerDependencies. We obtain the weekly download data by querying the npm registry API. This unlabelled dataset is used for the detection of zero-day vulnerabilities and the evaluation of false positives.
- Small-scale labeled dataset consisting of real-world, historically-vulnerable applications with CVE identifiers (called CVE Dataset). This dataset is compiled from the legacy Common Vulnerabilities and Exposures (CVEs) database and consists of 14 applications. In October 2023, we conducted an extensive keyword search on the National Vulnerability Database [23]. The search keywords include "react" along with a selection of React API names, including "dangerouslySetInnerHTML", "renderToStaticMarkup", "renderToString", and "useRef". We then study each vulnerability report along with its source code and exclude those not related to React. A list of the CVEs in this dataset is presented in Appendix B. This dataset—including XSS, arbitrary file upload, and improper authorization vulnerabilities—serves as ground truth for evaluating false negatives.

6.1.2 Experimental Environment. Our experiments are performed on a server with 64 GB memory, 16 Intel(R) Xeon(R) CPU E5-2620 v4 @ 2.10GHz cores with 2 threads per core, running Ubuntu 18.04.6 LTS. We run 16 processes of our system at the same time to speed up the analysis. Our baseline is a state-of-the-art static analysis tool, namely CodeQL [10], and we use their built-in CodeQL queries, including client-side cross-site scripting [9], stored cross-site scripting [32], and reflected cross-site scripting [28], for detecting application-level vulnerabilities and add our sources to CodeQL to detect package-level vulnerabilities. Note that our version of CodeQL is the one with their fix after we reported the problem of CodeQL in tracking React Data Flows to their developers [33].

6.2 RQ1: Zero-day Vulnerabilities

In this subsection, we answer the research question regarding the number of zero-day vulnerabilities detected by ReactAppScan but not existing approaches. Following prior works [59, 69], we consider a vulnerability as zero-day if it meets the following criteria: (i) it is not detected by prior work, such as CodeQL; (ii) there is no available information about the vulnerability, such as bug reports, CVE reports, or data in other vulnerability datasets based on our manual search; and (iii) it is validated through manual exploitation by a human expert. Note that in practice, when running on the large-scale unlabelled dataset, ReactAppScan only finds XSS vulnerabilities but not arbitrary file upload or improper authorization.

Table 3 shows a list of zero-day vulnerabilities detected by ReactAppScan on GitHub repositories and then Table 4 the list of zero-day vulnerabilities on NPM. Many of them are very popular, e.g., with more than 20K stars and 27K weekly downloads. In total, ReactAppScan detects 61 zero-day vulnerabilities with 13 on the application level and 48 on the package level from the large-scale dataset. Note that a single repository or package may contain more than one vulnerability. ReactAppScan outputs data flow paths and aggregates them by their last line of code. Paths ending on the same line of code are counted as one vulnerability.

A Case Study. We illustrate a case study using a zero-day vulnerability found by ReactAppScan. The vulnerability is located at rjsf-team/react-jsonschema-form [29], a 13,000-star GitHub repository for building JSON Schema [16] web forms. The corresponding

Table 3: A list of zero-day vulnerabilities detected by ReactAppScan in Github repositories.

Username/Repository	Tag/CommitId	Status	#Stars	#Vuls	Sinks
datopian/portaljs	f23d796	Reported	2,100+	3	setting innerHTML, <a href />
draft-js-plugins/draft-js-plugins	bae2bae	Reported	4,000+	1	<a href />
resendlabs/react-email	v0.0.14	Reported	11,000+	1	dangerouslySetInnerHTML
rjsf-team/react-jsonschema-form	v5.16.0	Acknowledged	13,000+	1	<a href />
plotly/dash	v2.14.2	Acknowledged	20,000+	1	<a href />
DimiMikadze/orca	53f761b	Fixed	1,200+	1	dangerouslySetInnerHTML
jonmircha/youtube-react	4946fb2	Reported	200+	1	dangerouslySetInnerHTML
Vagr9K/gatsby-advanced-starter	v4.17.0	Reported	1,600+	1	<a href />
unadlib/fronts	v0.1.1	Reported	500+	1	<iframe src />
virtualvivek/react-windows-ui	v4.2.2	Fixed	500+	1	<a href />
lucaspulliese/next-ecommerce	6c4888d	Reported	500+	1	dangerouslySetInnerHTML
justinmahar/react-social-media-embed	2d4e290	Reported	100+	2	<iframe src />, <a href />
aromalanil/markItDown	7d2fd34	Fixed	30+	1	dangerouslySetInnerHTML
ericclemmons/click-to-component	a9db3e1	Reported	1,500+	1	window.open
Aaditya1978/Bug-Blog	5027a83	Reported	10+	1	dangerouslySetInnerHTML
pramit-marattha/Fullstack-projects-frontend-with-react-and-backend-with-various-stacks	b4db8c2	Reported	160+	1	dangerouslySetInnerHTML
itsnitinr/driwwwle	782f64c	Fixed	120+	1	dangerouslySetInnerHTML
dunizb/CodeTest	81226bc	Reported	200+	1	dangerouslySetInnerHTML
refinedev/refine	5a3ad1d	Fixed	16,000+	1	location.replace
staringos/mtbird	d359c16	Fixed	400+	1	window.open
graphcommerce-org/graphcommerce	e534f170	Reported	200+	3	dangerouslySetInnerHTML
alibaba-fusion/materials	9658b8a	Reported	200+	1	<a href />
ice-lab/react-materials	65c5423	Reported	200+	1	dangerouslySetInnerHTML
gympass/yoga	dd4ef57	Reported	200+	1	<a href />
carbon-design-system/carbon-for-ibm-dotcom	f604b8c	Reported	200+	1	setting innerHTML
bangle-io/bangle-editor	45b40cf	Reported	600+	1	window.open
Muhammet-Yildiz/Mern-Blog	31d8569	Reported	40+	4	dangerouslySetInnerHTML
ant-design/pro-components	0e3609c	Reported	3,900+	1	dangerouslySetInnerHTML
nukeop/react-ui-cards	c0c75e5	Reported	200+	4	<a href />
rcaferati/react-awesome-button	a3954b9	Reported	1,200+	2	dangerouslySetInnerHTML

Table 4: A list of zero-day vulnerabilities detected by ReactAppScan in npm packages (19 in total).

Package	Version	Status	#Weekly Downloads	#Vuls
react-text-transition	1.3.0	Reported	27,000+	1
@hashicorp/react-hero	8.0.3	Reported	1,800+	2
@patternfly/react-docs	4.21.35	Reported	2,700+	1
@financial-times/dotcom-ui-header	2.6.2	Reported	3,000+	9
@hashicorp/react-consent-manager	7.1.0	Reported	2,300+	5
@financial-times/dotcom-ui-footer	2.7.2	Reported	2,900+	1

npm package, `@rjsf/core`, has 230,000 weekly downloads. The package provides a React component to build and customize web forms using JSON Schema. ReactAppScan reports a zero-day XSS vulnerability and the developers have acknowledged this vulnerability and are fixing it. Specifically, the package fails to adequately validate user input, resulting in adversary-controlled URLs being able to flow to the `<a href />` sink.

Figure 4 shows the simplified vulnerable code (Lines 6–22), along with its exploitation (Lines 2–4). The `FileWidget` component takes user input (Line 15) and generates a file download link that is controllable by an adversary (Line 8), leading to the XSS vulnerability. ReactAppScan successfully detects this vulnerability by tracing the data flow from `props` to the state (Line 17) and then across JSX attributes. In contrast, CodeQL fails to detect this vulnerability due to the extensive use of object destructuring with component props (Lines 6, 10, and 16), resulting in missing data flow edges.

```
1  // exploit
2  ReactDOM.render(
3    <FileWidget value={["javascript:alert(1)"]} options
         ={{ filePreview: true }} />
4  );
5  // code with vulnerability
6  function FileInfoPreview({ fileInfo }) {
7    const { dataURL, name } = fileInfo;
8    return <a download={`preview-${name}`} href={dataURL
         } />;
9  }
10 function FilesInfo({ filesInfo, preview }) {
11   return filesInfo.map((fileInfo) => {
12     return preview && <FileInfoPreview fileInfo={
           fileInfo} />;
13   });
14 }
15 function FileWidget(props) {
16   const { value, options } = props;
17   const [filesInfo, setFilesInfo] = useState(
18     Array.isArray(value) ? extractFileInfo(value) :
           extractFileInfo([value])
19   );
20   return <FilesInfo filesInfo={filesInfo} preview={
         options.filePreview} />;
21 }
22 export default FileWidget;
```

Figure 4: A Case Study of a Zero-day XSS Vulnerability in the rjsf-team/react-jsonschema-form GitHub Repository (13,000 stars). The vulnerability is acknowledged by the developers.

Table 5: A comparison of false discovery rate (FDR) and false negative rate (FNR) between ReactAppScan and CodeQL. FDR is evaluated on the large-scale dataset and FNR is evaluated on the CVE dataset. Note that both numbers are based on end-to-end, exploitable vulnerabilities.

Approach	FDR=FP/(FP+TP) ↓	FNR=FN/(FN+TP) ↓
ReactAppScan	15/96 (15.6%)	2/14 (14.2%)
CodeQL	72/94 (76.5%)	13/14 (92.8%)

6.3 RQ2: FP and FN

In this section, we evaluate the false positives and negatives of ReactAppScan in comparison with CodeQL using the large-scale and CVE datasets respectively. We inspect all detection results from the NPM dataset and all application-level results from the GitHub dataset. We only check package-level results from GitHub dataset that have over 200 stars. Table 5 shows an overview of the comparison, where ReactAppScan outperforms CodeQL in both FPs and FNs.

True Positives. Let us first discuss true positives detected by both ReactAppScan and CodeQL on both large-scale and CVE datasets. Note that a reported vulnerability is considered as true positive only if it is exploitable. First, on the large-scale dataset, CodeQL misses 61 true positives that are detected by ReactAppScan; as a comparison, ReactAppScan misses only two true positives detected by CodeQL. The main reason that ReactAppScan misses the vulnerabilities is the object explosion issue that leads to a scalability problem. Second, on the CVE dataset, ReactAppScan detected all vulnerabilities that are reported by CodeQL, while CodeQL misses 11 vulnerabilities detected by ReactAppScan.

False Positives. We conduct a manual inspection of detection results from ReactAppScan and CodeQL to evaluate False Positives, i.e., any vulnerability reporting from a detection tool that is *not* exploitable. We define the False Discovery Rate (FDR) as the ratio of FP to the sum of FP and TP, representing the proportion of reported vulnerabilities that are mistakenly identified. Note that a vulnerability is counted as a TP only if it can be exploited.

ReactAppScan has a much lower false discovery rate compared to CodeQL. We examine all the False Positives identified by ReactAppScan: The primary reason is due to the implementation of validation and data-flow sanitizations, making the detected vulnerabilities unexploitable. In contrast, CodeQL has a very high false discovery rate. This is mainly because of the overestimation of control and data flows in its syntax-driven approach. Besides, the predefined sources and sinks of CodeQL do not fit React.js applications perfectly. For example, its built-in queries only consider specific JSX attribute names, such as `dangerouslySetInnerHTML`, as sinks. This approach results in false positives when the JSX element is a `<script />`. Moreover, CodeQL analyzes all files in a repository, regardless of whether they are reachable or even dead code, leading to additional False Positives. In comparison, ReactAppScan starts from the application's entry point, which makes sure that vulnerabilities are at least reachable.

False Negatives. Our false negative evaluation is based on the ground truth information provided in the CVE dataset. ReactAppScan has two false negatives: (i) CVE-2023-34245 [7], attributable to unmodeled third-party libraries resulting in missing data flow, and (ii) CVE-2021-23398 [1], missed due to state explosion—specifically, a binary operation within a loop leading to timeout, which is a known limitation in existing JavaScript abstract interpretation [59, 69]. Note that there are additional FNs of ReactAppScan when we compare the TPs of ReactAppScan and CodeQL; however, since there is no ground truth information, it is challenging to measure FNR for the large-scale dataset.

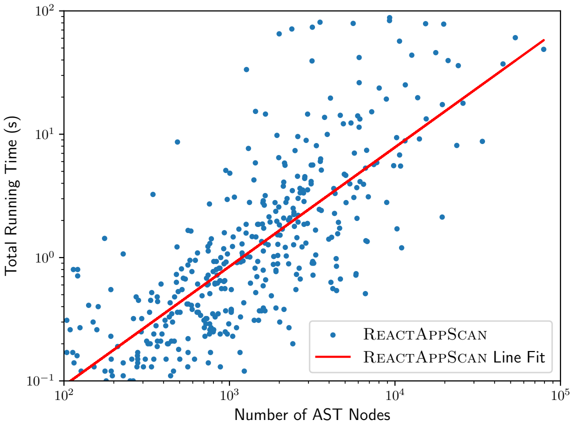


Figure 5: Total Running Time vs Number of AST Nodes for 500 random applications

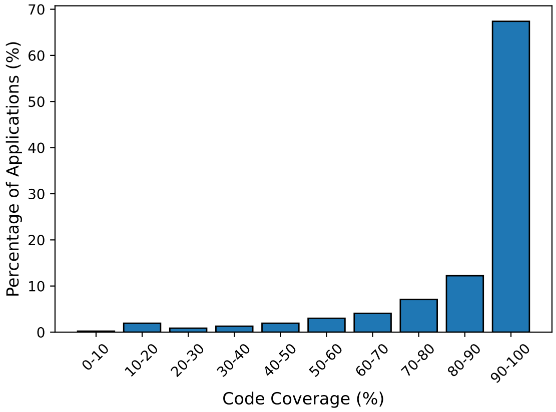


Figure 6: Code coverage distribution (500 random apps)

In contrast, CodeQL only detects one vulnerability in the CVE dataset. The main reason for CodeQL's bad performance is the incapability of tracking React data flows when functions are passed through JSX attributes across multiple components, as mentioned in our motivating example. Although we reported the issue to the developers, the fix only helped to detect one vulnerability. Additionally, dynamic JavaScript features, such as the propagation of JSX props using spread syntax and bracket syntax, also significantly contribute to CodeQL's bad performance in detecting CVE vulnerabilities.

6.4 RQ3: Performance

In this subsection, we answer the research question on the performance overhead and code coverage of ReactAppScan.

Analysis Time. We evaluate the total analysis time of ReactAppScan vs. the number of Abstract Syntax Tree (AST) Nodes for 500

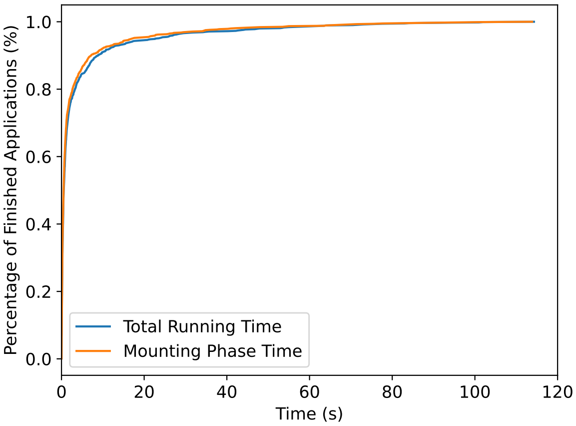


Figure 7: CDF of Analysis Time for 500 random applications

randomly selected applications from our large-scale dataset in Figure 5. When the number of AST nodes increases, the total running time increases linearly as we show the trend in a line fit. We also show a Cumulative Distribution Function (CDF) graph in Figure 7, which illustrates the total running time with a 120-second time-out threshold. ReactAppScan completes the analysis of 95% of the applications within 30 seconds, and 97% within 60 seconds. This indicates the high efficiency of ReactAppScan in processing a significant majority of React packages. The total running time closely aligns with the duration of the mounting phase, suggesting small performance overhead during the updating and unmounting phase.

Code Coverage. We evaluate statement coverage, defined as the percentage of statements executed by ReactAppScan, i.e., the number of analyzed statements divided by the total. Note that our measurement methodology and tooling are inherited from prior work [69], which covers all the statements within an application, including both client-side and server-side codes. This metric demonstrates how complete our system is in analyzing React applications. Figure 6 presents a distribution graph of statement coverage when analyzing 500 randomly selected React applications, each with a timeout of 120 seconds. In our evaluation, 67.3% of the React applications have 100% statement coverage. This number surpasses ODGen's code coverage, where only about 40% of applications reach 100% statement coverage. The higher code coverage of ReactAppScan compared to ODGen can be attributed to the less common practice in client-side React applications of dynamically including files based on input, a scenario that cannot be statically resolved. While React does allow for dynamic imports [18], the paths used in React applications are typically predefined.

7 DISCUSSION

Ethics: Responsible Disclosure. We have responsibly disclosed all zero-day vulnerabilities found by ReactAppScan to their developers together with suggested fixes via either emails, GitHub issues or pull requests. So far, six vulnerabilities have already been fixed and two have been acknowledged and under fixing.

General Single-page Application. React is one single-page application framework and there are others, such as Angular.js. The high-level idea of component graph applies to other single-page applications because components are also used by other frameworks, such as Angular.js, to model Unidirectional Data Flows. At the same time, our current implementation only supports React, because Angular.js heavily relies on TypeScript. We will leave those as our future work to support other single-page application frameworks.

Analysis Soundness. Our analysis is unsound, which is the same as all prior abstract interpretation works [59, 69, 90]. There are different reasons for unsoundness. First, JavaScript may introduce dynamic code via function calls, such as `eval` and `new Function`. ReactAppScan, just like all prior works, may not resolve such dynamically-introduced code especially when it is related to user inputs. Second, ReactAppScan overestimates database-related dependencies by only checking for common keys between query filters and model properties using a regular expression, especially for those queries that affect multiple keys or entries. Third, the URL matching mechanism for client-server data dependencies can fail to find a match, such as when there is an unresolved variable from user input in the URL, leading to potential false negatives. Lastly, the current implementation fully supports React features up to version 16 for React data flows. That is, new or experimental features from newer versions like version 18 may lead to unsoundness.

State Explosion. ReactAppScan, being similar to existing abstract interpretation [59, 69, 90], may have the problem of state explosion, especially for heavily-embedded branching statements or ternary operators. At the same time, the percentage of state explosion is relatively smaller compared with general NPM packages: For example, ReactAppScan only encounters one example in the CVE dataset, which suffers from state explosion. The reason might be different coding practices for React and general NPM developers.

Execution Order of Asynchronous Events: Theoretically, asynchronous events, e.g., React lifecycle events, can happen in different orders, but ReactAppScan only abstractly interprets them in one particular order following the sequence in the queue. This can lead to both FPs and FNs. Note that we would expect that FPs are rare because events can usually happen in any order. Similarly, FNs are rare too, because even if the order is different, two pieces of dataflows are still established and ReactAppScan can find a path.

Analysis of Transpiled JSX Code. One possible solution of JSX analysis and vulnerability detection is to transpile JSX code to JavaScript and apply state-of-the-art JavaScript analysis [59, 69, 90]. However, such an approach is not scalable, and will significantly suffer from the problem of state explosion. Specifically, according to our experiments, neither ODGen [69] nor FAST [59] can finish analyzing the transpiled code of a simple demo application let alone those applications in the large-scale or CVE database. In addition, the analysis of transpiled code will lose the JSX syntax and their information, such as React dataflow. This is similar to the comparison of binary vs. source code analysis. Although binary analysis is available, source code analysis will also preserve more information and greatly improve the analysis accuracy.

8 RELATED WORK

React Security. React implements many built-in security features to defend against various possible attacks. For example, React escapes any values embedded in JSX by default [17], thereby preventing injection attacks. Despite these built-in features, due to the functionality reason, React also includes `dangerouslySetInnerHTML` [11], which can bypass this escaping mechanism and is also considered as sinks in our work. To the best of our knowledge, prior work on React vulnerability detection is limited. CodeQL [10], an industry-level analysis engine for semantics-based search on a target codebase, provides standard libraries for data flow analysis and for working with React. React developer tool [26], although capable of analyzing React applications dynamically, is only used for performance profiling but not vulnerability detection.

Static Analysis for JavaScript. In the past, there have been many static analysis works that were proposed for different purposes, such as type inference. TAJS [57] abstractly interpret JavaScript programs to infer type information and detect programming errors. Similarly, JSAI [61] uses abstract interpretation for JavaScript type inference, pointer analysis, and control-flow analysis. SAFE [66] and SAFEWAPI [37] covert JavaScript to an Intermediate Representation for abstract interpretation. Zheng et al. [93] propose a static analysis method to detect non-deterministic problems caused by asynchronous AJAX calls. Madsen et al. [70] present an event-based call graph to detect bugs related to event handling in Node.js applications. AdGraph [55] represents interactions between HTML structure, network requests, and JavaScript behavior. As a comparison, prior static analysis focuses on JavaScript instead of JSX and React and there are challenges in analyzing JSX, such as React data flows between components.

Detection of Node.js Vulnerability. In the past, researchers have studied various security issues of Node.js, e.g., supply chain security [46, 82], Regular Expression Denial of Service (ReDoS) [38, 45, 80], privilege reduction [82], debloating [65], hidden property abuse [88], and prototype pollution [60, 63, 79]. The techniques in detecting Node.js vulnerabilities also range from static analysis to dynamic analysis. We start with dynamic analysis. Jalangi [78] dynamically analyzes JavaScript applications with selective record-replay, shadow values and shadow execution. Arteau [35] detects prototype pollution vulnerabilities with a dynamic fuzzer. We then describe existing static analysis in detecting Node.js vulnerabilities. DAPP [64] detects prototype pollution vulnerabilities based on abstract syntax tree and control flow graph. Several works, such as ObjLupAnsys [68], ODGen [69], CoCo [90], and Nodest [73], detect JavaScript vulnerabilities using abstract interpretation. Node.js ecosystem security is also studied. ConflictJS [76] analyzes Node.js libraries to find conflicts. Zimmermann et al. [94] studies security risks of third-party Node.js dependencies. NodeMedic [44] proposes provenance graph to detect vulnerabilities in Node.js packages. Brown et al. [39] study security problems in the binding layers of Node.js. As a comparison, ReactAppScan's objective is to detect React vulnerabilities, i.e., out of scope of these prior works.

Client-side JavaScript Security The detection and prevention of client-side cross-site scripting (XSS) [67, 71, 72, 81, 83] have been well-studied in the past. Prior work proposes preventing XSS attacks via Content Security Policy (CSP), e.g., CSPAutoGen [75]. Pathcutter [43] cuts off the propagation path of XSS worms through view separation. Zhang et al. [91] develop a browser-based framework for analyzing code integrity problems caused by JavaScript global identifier conflicts. JSIsolate [92], provides a browser-based, isolated, and reliable JavaScript execution environment based on the dependency relationship of different JavaScript program components. Browser fingerprinting [41, 54, 86, 87] and web tracking [74] have also been studied by researchers. Deemon [77] is a framework for detecting CSRF vulnerabilities with a unified property graph built with dynamic traces. Melicher et al. [71] and Steffens et al. [81] adopt dynamic taint analysis to find DOM-based XSS. HideNoSeek [48], JShield [42], JaSt [50], and JStap [49] study detecting and defending against malicious client-side JavaScript programs. Black Window [47] is a black box data-driven approach to web crawling and scanning for finding cross-site scripting vulnerabilities. Jin et al. [58] propose a DOM-tree type, a predefined set of expected DOM trees for Electron apps, to defend against unintended DOM-tree mutations at runtime. As a comparison, ReactAppScan does not require dynamic analysis. Moreover, none of these methods track data flow in React or cross-side data dependencies.

Graph-based Vulnerability Detection. Program analysis, especially graph-based analysis, is heavily used for security analysis, especially vulnerability detection. Yamaguchi et al. [89] propose Code Property Graph (CPG), a joint data structure of abstract syntax trees, control flow graphs and program dependence graph, to detect vulnerabilities with graph traversals. Backes et al. [36] extends CPG with call graphs for PHP vulnerability detection. Jensen et al. [56] utilize static analysis for detecting both dataflow-related and type-related programming errors in browser-based JavaScript applications, which models both the DOM model of the browser API and HTML page. JAW [62] introduces the Hybrid Property Graph, a code representation that includes Event Registration, Dispatch, and Dependency Graph to capture event-based transfer of control. Taintmini [85] is a static taint analysis method designed to detect the flow of sensitive data in mini-programs. DoubleX [51] introduces Extension Dependence Graph (EDG) to detect vulnerabilities in browser extensions. As a comparison, from a high-level, ReactAppScan is also a graph-based analysis, but ReactAppScan focuses on the detection of React application vulnerabilities.

9 CONCLUSION

Single-page application frameworks, such as React, have recently become popular and widely used by many top websites and web applications. At the same time, vulnerability detection for React applications falls behind: Many vulnerability detection approaches do not support React applications, and those that support React also fall short in modeling React data flows, leading to the incapability of detecting many real-world React application vulnerabilities.

In this paper, we design a novel, *open-source* vulnerability detection system, called ReactAppScan, which models React components as Component Graph with data flows among their props and states. ReactAppScan builds the component graph via abstract interpretation with monitoring of state and props change and then performs graph queries to mine vulnerabilities. Our evaluation shows that ReactAppScan detected 61 zero-day vulnerabilities; we have reported all of them to their developers and so far six have already been fixed. We also compare ReactAppScan with CodeQL, the state-of-the-art approach in detecting React application vulnerabilities, and show that ReactAppScan significantly outperforms CodeQL with much lower false positive and negative rates.

ACKNOWLEDGMENTS

We would like to thank anonymous reviewers for their helpful comments and feedback. This work was supported in part by National Science Foundation (NSF) under grants CNS-21-54404 and CNS-20-46361 and awards 2330565 and 1918542, a Defense Advanced Research Projects Agency (DARPA) Young Faculty Award (YFA) under Grant Agreement D22AP00137-00, the UK Research and Innovation organization (UKRI) under award EP/Y026233/1, an Amazon Research Award (ARA) 2021, and gifts from Visa Research. 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, DARPA, Amazon, Visa Research, or UKRI.

REFERENCES

[1] 2021. CVE-2021-23398 Detail. Retrieved Jan 6, 2024 from https://nvd.nist.gov/vuln/detail/CVE-2021-23398
[2] 2021. CVE-2021-31712 Detail. Retrieved Jan 6, 2024 from https://nvd.nist.gov/vuln/detail/CVE-2021-31712
[3] 2022. *JSX*. Retrieved Dec 21, 2023 from https://facebook.github.io/jsx/
[4] 2022. ODGen. Retrieved Nov 20, 2023 from https://github.com/Song-Li/ODGen
[5] 2023. 2023 Developer Survey. Retrieved Jan 10, 2024 from https://survey.stackoverflow.co/2023/#section-most-popular-technologies-web-frameworks-and-technologies
[6] 2023. CVE-2023-22462 Detail. Retrieved Jan 6, 2024 from https://nvd.nist.gov/vuln/detail/CVE-2023-22462
[7] 2023. CVE-2023-34245 Detail. Retrieved Jan 6, 2024 from https://nvd.nist.gov/vuln/detail/CVE-2023-34245
[8] 2023. CVE-2023-5654 Detail. Retrieved Jan 6, 2024 from https://nvd.nist.gov/vuln/detail/CVE-2023-5654
[9] 2024. Client-side cross-site scripting. Retrieved Jan 5, 2024 from https://codeql.github.com/codeql-query-help/javascript/js-xss/
[10] 2024. CodeQL. Retrieved Jan 6, 2024 from https://codeql.github.com/
[11] 2024. Dangerously setting the inner HTML. https://react.dev/reference/react-dom/components/common#dangerously-setting-the-inner-html
[12] 2024. DOMPurify - a DOM-only, super-fast, uber-tolerant XSS sanitizer for HTML, MathML and SVG. https://github.com/cure53/DOMPurify
[13] 2024. *Espree*. https://github.com/eslint/espree.
[14] 2024. Express - Node.js web application framework. Retrieved Jan 19, 2024 from https://expressjs.com/
[15] 2024. HTML 5. Retrieved Jan 19, 2024 from https://www.w3.org/TR/2008/WD-html5-20080610/dom.html#innerhtml0
[16] 2024. JSON Schema. https://json-schema.org/
[17] 2024. JSX Prevents Injection Attacks. Retrieved Jan 10, 2024 from https://legacy.reactjs.org/docs/introducing-jsx.html#jsx-prevents-injection-attacks
[18] 2024. lazy. Retrieved Jan 6, 2024 from https://react.dev/reference/react/lazy
[19] 2024. Managing State. https://react.dev/learn/managing-state
[20] 2024. markdown-it - Markdown parser, done right. https://github.com/markdown-it/markdown-it/tree/master
[21] 2024. MongoDB: The Developer Data Platform. Retrieved Jan 19, 2024 from https://www.mongodb.com/
[22] 2024. Mongoose: elegant mongodb object modeling for node.js. https://mongoosejs.com/
[23] 2024. National Vulnerability Database. Retrieved Jan 5, 2024 from https://nvd.nist.gov/
[24] 2024. Passing Props to a Component. Retrieved Jan 19, 2024 from https://react.dev/learn/passing-props-to-a-component
[25] 2024. React. Retrieved Jan 6, 2024 from https://react.dev/
[26] 2024. React Developer Tools. https://react.dev/learn/react-developer-tools
[27] 2024. ReactAppScan Open-Source Repository. https://github.com/react-app-scan/react-app-scan
[28] 2024. Reflected cross-site scripting. Retrieved Jan 5, 2024 from https://codeql.github.com/codeql-query-help/javascript/js-reflected-xss/
[29] 2024. rjsf-team/react-jsonschema-form. https://github.com/rjsf-team/react-jsonschema-form
[30] 2024. sanitize-html. https://www.npmjs.com/package/sanitize-html
[31] 2024. State: A Component's Memory. Retrieved Jan 19, 2024 from https://react.dev/learn/state-a-components-memory
[32] 2024. Stored cross-site scripting. Retrieved Jan 5, 2024 from https://codeql.github.com/codeql-query-help/javascript/js-stored-xss/
[33] 2024. Taint Tracking of Function Passed Through JSX Attributes. https://github.com/github/codeql/issues/15207.
[34] 2024. Writing Markup with JSX. Retrieved Jan 6, 2024 from https://react.dev/learn/writing-markup-with-jsx/
[35] Olivier Arteau. 2018. Prototype pollution attack in nodejs application.
[36] Michael Backes, Konrad Rieck, Malte Skoruppa, Ben Stock, and Fabian Yamaguchi. 2017. Efficient and Flexible Discovery of PHP Application Vulnerabilities. In *2017 IEEE European Symposium on Security and Privacy (EuroS&P)*. 334–349. https://doi.org/10.1109/EuroSP.2017.14
[37] SungGyeong Bae, Hyunghun Cho, Inho Lim, and Sukyoung Ryu. 2014. SAFEWAPI: web API misuse detector for web applications. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Hong Kong, China) *(FSE 2014)*. New York, NY, USA, 507–517.
[38] Zhihao Bai, Ke Wang, Hang Zhu, Yinzhi Cao, and Xin Jin. 2021. Runtime recovery of web applications under zero-day redos attacks. In *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1575–1588.
[39] Fraser Brown, Shravan Narayan, Riad S. Wahby, Dawson Engler, Ranjit Jhala, and Deian Stefan. 2017. Finding and Preventing Bugs in JavaScript Bindings. In *2017 IEEE Symposium on Security and Privacy (SP)*. 559–578.
[40] BuiltWith. [n. d.]. React Usage Statistics. Retrieved Jan 18, 2024 from https://trends.builtwith.com/javascript/React
[41] Yinzhi Cao, Song Li, and Erik Wijmans. 2017. (Cross-) browser fingerprinting via OS and hardware level features. In *Proceedings 2017 Network and Distributed System Security Symposium*. Internet Society.
[42] Yinzhi Cao, Xiang Pan, Yan Chen, and Jianwei Zhuge. 2014. JShield: towards real-time and vulnerability-based detection of polluted drive-by download attacks. In *Proceedings of the 30th Annual Computer Security Applications Conference* (New Orleans, Louisiana, USA) *(ACSAC '14)*. New York, NY, USA, 466–475.
[43] Yinzhi Cao, Vinod Yegneswaran, Phillip A. Porras, and Yan Chen. 2012. PathCutter: Severing the Self-Propagation Path of XSS JavaScript Worms in Social Web Networks. In *Network and Distributed System Security Symposium*.
[44] Darion Cassel, Wai Tuck Wong, and Limin Jia. 2023. NodeMedic: End-to-End Analysis of Node.js Vulnerabilities with Provenance Graphs. In *2023 IEEE 8th European Symposium on Security and Privacy (EuroS&P)*. 1101–1127.
[45] James C Davis, Eric R Williamson, and Dongyoon Lee. 2018. A Sense of Time for JavaScript and Node.js: First-Class Timeouts as a Cure for Event Handler Poisoning. In *27th USENIX Security Symposium (USENIX Security 18)*. 343–359.
[46] Ruian Duan, Omar Alrawi, Ranjita Pai Kasturi, Ryan Elder, Brendan Saltaformaggio, and Wenke Lee. 2020. Towards measuring supply chain attacks on package managers for interpreted languages. *arXiv preprint arXiv:2002.01139* (2020).
[47] Benjamin Eriksson, Giancarlo Pellegrino, and Andrei Sabelfeld. 2021. Black Widow: Blackbox Data-driven Web Scanning. In *2021 IEEE Symposium on Security and Privacy (SP)*. 1125–1142. https://doi.org/10.1109/SP40001.2021.00022
[48] Aurore Fass, Michael Backes, and Ben Stock. 2019. HideNoSeek: Camouflaging Malicious JavaScript in Benign ASTs. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security* (London, United Kingdom) *(CCS '19)*. Association for Computing Machinery, New York, NY, USA, 1899–1913.
[49] Aurore Fass, Michael Backes, and Ben Stock. 2019. JStap: a static pre-filter for malicious JavaScript detection. In *Proceedings of the 35th Annual Computer Security Applications Conference* (San Juan, Puerto Rico, USA) *(ACSAC '19)*. Association for Computing Machinery, New York, NY, USA, 257–269.
[50] Aurore Fass, Robert P. Krawczyk, Michael Backes, and Ben Stock. 2018. JaSt: Fully Syntactic Detection of Malicious (Obfuscated) JavaScript. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, Cristiano Giuffrida, Sébastien Bardin, and Gregory Blanc (Eds.). Cham, 303–325.
[51] Aurore Fass, Dolière Francis Somé, Michael Backes, and Ben Stock. 2021. DoubleX: Statically Detecting Vulnerable Data Flows in Browser Extensions at Scale. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security* (Virtual Event, Republic of Korea) *(CCS '21)*. 1789–1804.
[52] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. 1987. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.* 9, 3 (jul 1987), 319–349. https://doi.org/10.1145/24039.24041
[53] Veronica Gavrilă, Lidia Băjenaru, and Ciprian Dobre. 2019. Modern single page application architecture: a case study. *Stud. Inform. Control* 28 (2019), 231–238.
[54] Alejandro Gómez-Boix, Pierre Laperdrix, and Benoit Baudry. 2018. Hiding in the crowd: an analysis of the effectiveness of browser fingerprinting at large scale. In *Proceedings of the 2018 world wide web conference*. 309–318.
[55] Umar Iqbal, Peter Snyder, Shitong Zhu, Benjamin Livshits, Zhiyun Qian, and Zubair Shafiq. 2020. Adgraph: A graph-based approach to ad and tracker blocking. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 763–776.
[56] Simon Holm Jensen, Magnus Madsen, and Anders Møller. 2011. Modeling the HTML DOM and browser API in static analysis of JavaScript web applications. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. 59–69.
[57] Simon Holm Jensen, Anders Møller, and Peter Thiemann. 2009. Type Analysis for JavaScript. In *Static Analysis*, Jens Palsberg and Zhendong Su (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 238–255.
[58] Zihao Jin, Shuo Chen, Yang Chen, Haixin Duan, Jianjun Chen, and Jianping Wu. 2023. A Security Study about Electron Applications and a Programming Methodology to Tame DOM Functionalities. In *NDSS*.

[59] Mingqing Kang, Yichao Xu, Song Li, Rigel Gjomemo, Jianwei Hou, V. N. Venkatakrishnan, and Yinzhi Cao. 2023. Scaling JavaScript Abstract Interpretation to Detect and Exploit Node.js Taint-style Vulnerability. In *2023 IEEE Symposium on Security and Privacy (SP)*. 1059–1076.

[60] Zifeng Kang, Song Li, and Yinzhi Cao. 2022. Probe the Proto: Measuring Client-Side Prototype Pollution Vulnerabilities of One Million Real-world Websites. In *Network and Distributed System Security Symposium (NDSS 2022)*.

[61] Vineeth Kashyap, Kyle Dewey, Ethan A. Kuefner, John Wagner, Kevin Gibbons, John Sarracino, Ben Wiedermann, and Ben Hardekopf. 2014. JSAI: a static analysis platform for JavaScript. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2014)*. 121–132.

[62] Soheil Khodayari and Giancarlo Pellegrino. 2021. JAW: Studying Client-side CSRF with Hybrid Property Graphs and Declarative Traversals. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, 2525–2542. https://www.usenix.org/conference/usenixsecurity21/presentation/khodayari

[63] Hee Yeon Kim, Ji Hoon Kim, Ho Kyun Oh, Beom Jin Lee, Si Woo Mun, Jeong Hoon Shin, and Kyounggon Kim. 2022. DAPP: automatic detection and analysis of prototype pollution vulnerability in Node.js modules. *International Journal of Information Security* 21, 1 (2022), 1–23.

[64] Hee Yeon Kim, Ji Hoon Kim, Ho Kyun Oh, Beom Jin Lee, Si Woo Mun, Jeong Hoon Shin, and Kyounggon Kim. 2022. DAPP: automatic detection and analysis of prototype pollution vulnerability in Node.js modules. *Int. J. Inf. Secur.* 21, 1 (feb 2022), 1–23.

[65] Igibek Koishybayev and Alexandros Kapravelos. 2020. Mininode: Reducing the attack surface of Node.js applications. In *23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020)*. 121–134.

[66] Hongki Lee, Sooncheol Won, Joonho Jin, Junhee Cho, and Sukyoung Ryu. 2012. SAFE: Formal specification and implementation of a scalable analysis framework for ECMAScript. In *FOOL 2012: 19th International Workshop on Foundations of Object-Oriented Languages*. Citeseer, 96.

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

[68] Song Li, Mingqing Kang, Jianwei Hou, and Yinzhi Cao. 2021. 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)*. 268–279.

[69] Song Li, Mingqing Kang, Jianwei Hou, and Yinzhi Cao. 2022. Mining Node.js Vulnerabilities via Object Dependence Graph and Query. In *31st USENIX Security Symposium (USENIX Security 22)*. Boston, MA, 143–160.

[70] Magnus Madsen, Frank Tip, and Ondřej Lhoták. 2015. Static analysis of event-driven Node.js JavaScript applications. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2015)*. 505–519.

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

[72] Yacin Nadji, Prateek Saxena, and Dawn Song. 2009. Document Structure Integrity: A Robust Basis for Cross-site Scripting Defense.. In *NDSS*, Vol. 20.

[73] Benjamin Barslev Nielsen, Behnaz Hassanshahi, and François Gauthier. 2019. 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* (Tallinn, Estonia) *(ESEC/FSE 2019)*. 455–465.

[74] Xiang Pan, Yinzhi Cao, and Yan Chen. 2015. 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*.

[75] Xiang Pan, Yinzhi Cao, Shuangping Liu, Yu Zhou, Yan Chen, and Tingzhe Zhou. 2016. CSPAutoGen: Black-box Enforcement of Content Security Policy upon Real-world Websites. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (Vienna, Austria) *(CCS '16)*. 653–665.

[76] Jibesh Patra, Pooja N. Dixit, and Michael Pradel. 2018. ConflictJS: Finding and Understanding Conflicts Between JavaScript Libraries. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. 741–751.

[77] Giancarlo Pellegrino, Martin Johns, Simon Koch, Michael Backes, and Christian Rossow. 2017. Deemon: Detecting CSRF with dynamic analysis and property graphs. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 1757–1771.

[78] Koushik Sen, Swaroop Kalasapur, Tasneem Brutch, and Simon Gibbs. 2013. Jalangi: a selective record-replay and dynamic analysis framework for JavaScript. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering* (Saint Petersburg, Russia) *(ESEC/FSE 2013)*. 488–498.

[79] Mikhail Shcherbakov, Musard Balliu, and Cristian-Alexandru Staicu. 2023. Silent Spring: Prototype Pollution Leads to Remote Code Execution in Node.js. *USENIX Security*.

[80] Cristian-Alexandru Staicu and Michael Pradel. 2018. Freezing the Web: A Study of ReDoS Vulnerabilities in JavaScript-based Web Servers. In *27th USENIX Security Symposium (USENIX Security 18)*. 361–376.

Table 6: A List of vulnerabilities used in our CVE dataset

Vulnerability Type	CVE#
Cross-site Scripting (XSS)	CVE-2023-41167, CVE-2023-37259, CVE-2023-34245, CVE-2023-30609, CVE-2023-22462, CVE-2023-25572, CVE-2021-23398, CVE-2021-31712, CVE-2020-12113, CVE-2021-41249, CVE-2020-15119
Improper Authorization	CVE-2023-5654
Unrestricted File Upload	CVE-2021-32622
Insufficient Data Authenticity	CVE-2021-21320

[81] Marius Steffens, Christian Rossow, Martin Johns, and Ben Stock. 2019. Don't Trust The Locals: Investigating the Prevalence of Persistent Client-Side Cross-Site Scripting in the Wild. (2019).

[82] Nikos Vasilakis, Cristian-Alexandru Staicu, Grigoris Ntousakis, Konstantinos Kallas, Ben Karel, André DeHon, and Michael Pradel. 2021. Preventing dynamic library compromise on Node.js via rwx-based privilege reduction. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. 1821–1838.

[83] Philipp Vogt, Florian Nentwich, Nenad Jovanovic, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. 2007. Cross site scripting prevention with dynamic data tainting and static analysis.. In *NDSS*, Vol. 2007. 12.

[84] W3Techs. [n. d.]. Historical trends in the usage statistics of React versions for websites. Retrieved Jan 10, 2024 from https://w3techs.com/technologies/history_details/js-react

[85] Chao Wang, Ronny Ko, Yue Zhang, Yuqing Yang, and Zhiqiang Lin. 2023. Taintmini: Detecting Flow of Sensitive Data in Mini-Programs with Static Taint Analysis. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. 932–944. https://doi.org/10.1109/ICSE48619.2023.00086

[86] Shujiang Wu, Song Li, Yinzhi Cao, and Ningfei Wang. 2019. Rendered private: Making {GLSL} execution uniform to prevent {WebGL-based} browser fingerprinting. In *28th USENIX Security Symposium (USENIX Security 19)*. 1645–1660.

[87] Shujiang Wu, Pengfei Sun, Yao Zhao, and Yinzhi Cao. 2023. Him of many faces: Characterizing billion-scale adversarial and benign browser fingerprints on commercial websites. In *30th Annual Network and Distributed System Security Symposium, NDSS*.

[88] Feng Xiao, Jianwei Huang, Yichang Xiong, Guangliang Yang, Hong Hu, Guofei Gu, and Wenke Lee. 2021. Abusing hidden properties to attack the Node.js ecosystem. In *30th USENIX Security Symposium (USENIX Security 21)*. 2951–2968.

[89] Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. 2014. Modeling and Discovering Vulnerabilities with Code Property Graphs. In *2014 IEEE Symposium on Security and Privacy*. 590–604. https://doi.org/10.1109/SP.2014.44

[90] Jianjia Yu, Song Li, Junmin Zhu, and Yinzhi Cao. 2023. 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)*. 2441–2455.

[91] Mingxue Zhang and Wei Meng. 2020. Detecting and understanding JavaScript global identifier conflicts on the web. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2020)*. 38–49.

[92] Mingxue Zhang and Wei Meng. 2021. JSISOLATE: lightweight in-browser JavaScript isolation. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Athens, Greece) *(ESEC/FSE 2021)*. Association for Computing Machinery, New York, NY, USA, 193–204.

[93] Yunhui Zheng, Tao Bao, and Xiangyu Zhang. 2011. Statically locating web application bugs caused by asynchronous calls. In *Proceedings of the 20th International Conference on World Wide Web* (Hyderabad, India) *(WWW '11)*. Association for Computing Machinery, New York, NY, USA, 805–814.

[94] Markus Zimmermann, Cristian-Alexandru Staicu, Cam Tenny, and Michael Pradel. 2019. Small world with high risks: A study of security threats in the npm ecosystem. In *28th USENIX Security Symposium (USENIX Security 19)*. 995–1010.

Appendices

A OPERATIONAL SEMANTICS

Figure 8 depicts the detailed operational semantics.

B A LIST OF ZERO-DAY VULNERABILITIES

Table 6 shows a list of React vulnerabilities and their CVE identifiers in our CVE dataset.

Phase I: Mounting (JSX Elements)

$$\frac{p \Rightarrow (N, E, el, q, S),\ e_1 \Rightarrow (N_{e_1}, E_{e_1}, el, q, S)}{((\mathit{ElName}\ x, \mathit{Attrs}\ e_1), a, p) \Rightarrow (N \cup N_{e_1}, E \cup E_{e_1} \cup \mathit{AddEdge}^{el \to el}_{el \to el_{\texttt{new}}}, el_{\texttt{new}}, q, S),\ \texttt{where}\ el_{\texttt{new}} := \mathit{AddEl}^{a}_{a.x.\texttt{name}}} \quad \text{(JSXOpeningElement)}$$

$$\frac{p \Rightarrow (N, E, el, q, S, (e_1, a.e_1, p) \Rightarrow (N_{e_1}, E_{e_1}, el_{e_1}, q, S), \ldots, (e_n, a.e_n, p, q) \Rightarrow (N_{e_n}, E_{e_n}, el_{e_n}, q, S)}{((\mathit{Child}\ e_1, \ldots, \mathit{Child}\ e_n), a, p, q) \Rightarrow \left(\bigcup_{i=1}^{n} N_{e_i}, \bigcup_{i=1}^{n} E_{e_i} \cup \bigcup_{i=1}^{n} \texttt{AddEdge}^{el \to el}_{el \to e_i}, el, q, S\right)} \quad \text{(JSXChildren)}$$

Phase I: Mounting (JSX Attributes and Props)

$$\frac{p \Rightarrow (N, E, el, q, S), (e_1, a.e_1, p) \Rightarrow (N_{e_1}, E_{e_1}, el, q, S), (e_2, a.e_2, p) \Rightarrow (N_{e_2}, E_{e_2}, el, q, S)}{((\texttt{name}\ e_1 = \texttt{Value}\ e_2), a, p) \Rightarrow\ (N \cup N_{e_1} \cup N_{e_2}, E \cup E_{e_1} \cup E_{e_2} \cup E_{\texttt{attr}} \cup E_{\texttt{props}}, el, q, S)}\ \texttt{where} \begin{cases} E_{\texttt{attr}} := \texttt{AddEdge}^{\texttt{attr} \to o}_{\texttt{attr}' \to o'}, \forall o' \in \texttt{Child}^{a \to o}_{a.e2}, \texttt{attr}' = \texttt{LkupAttr}(a.e_1) \\ E_{\texttt{props}} := \texttt{AddProperty}^{\texttt{props} \to o'}_{a.e_1.name}, \texttt{props} := \texttt{LkupPropsObjs}(el)\ ,\ el \in N_c \end{cases} \quad \text{(JSXAttribute)}$$

$$\frac{p \Rightarrow (N, E, el, q, S), (e_1, a_{e_1}, p) \Rightarrow (N_{e_1}, E_{e_1}, el, q, S), \ldots, (e_n, a_{e_n}, p, q) \Rightarrow (N_{e_n}, E_{e_n}, el, q, S)}{((\texttt{Attr}\ e_1, \ldots, \texttt{Attr}\ e_n), a, p) \Rightarrow (\bigcup_{i=1}^{n} N_{e_i}, \bigcup_{i=1}^{n} E_{e_i}, el, q, S)} \quad \text{(JSXAttributes)}$$

$$\frac{p \Rightarrow (N, E, el, q, S), (e, a_e, p) \Rightarrow (N_e, E_e, el, q, S), r := \texttt{AddNode}^{o}_{a}, c := \texttt{AddNode}^{o}_{a}, p := \texttt{AddProperty}^{r \to o}_{\texttt{current}}}{(\mathit{useRef}(e), a, p) \Rightarrow (N \cup r \cup c, E \cup p, el, q, S)} \quad \text{(useRef)}$$

Phase I: Mounting (JSX State)

$$\frac{p \Rightarrow (N, E, el, q, S), (e, a.e, p) \Rightarrow (N_e, E_e, el, q, S)}{(\mathit{useState}(e), a, p) \Rightarrow \texttt{if LkupState}(el) \neq \emptyset\ \text{then}\ (N, E, el, q)\ \texttt{else}\ (N \cup N_e \cup N_{\texttt{state}} \cup N_{\texttt{state\_v}} \cup N_{\texttt{setState}}, E \cup E_e \cup E_{\texttt{state}} \cup E_{\texttt{setState}} \cup E_v, el, q, S)}$$

$$\texttt{where} \begin{cases} N_{\texttt{state}} := \texttt{AddNode}^{\texttt{state}}_{a} \\ N_{\texttt{state\_v}} := \texttt{AddNode}^{v}_{a} \\ N_{\texttt{setState}} := \texttt{AddNode}^{v}_{a} \end{cases} \& \begin{cases} E_{\texttt{state}} := \texttt{AddEdge}^{c \to \texttt{state}}_{el \to N_{\texttt{state}}} \\ E_{\texttt{setState}} := \texttt{AddEdge}^{\texttt{state} \to <v, v_f>}_{N_{\texttt{state}} \to <N_{\texttt{state\_v}}, N_{\texttt{setState}}>} \\ E_v := \texttt{AddEdge}^{v \to o}_{N_{\texttt{state\_v}} \to o'}, \forall o' \in \texttt{Child}^{a \to o}_{a.e} \end{cases} \quad \text{(useState)}$$

Phase I: Mounting (Component Rendering)

$$\frac{p \Rightarrow (N, E, el, q, S), e_1 \Rightarrow (N_{e_1}, E_{e_1}, el_{e_1}, q, S), e_2 \Rightarrow (N_{e_2}, E_{e_2}, el_{e_1}, q, S)}{((\mathit{OpeningEl}\ e_1, \mathit{Children}\ e_2), a, p) \Rightarrow (N \cup N_{e_1} \cup N_{e_2} \cup N_r, E \cup E_{e_1} \cup E_{e_2} \cup E_r), el_{e_1}, q \cup q_u, S \cup \{el_{e1} :< \texttt{LkupStateObjs}(el_{e1}), \texttt{LkupPropsObjs}(el_{e1})\ >\})}$$

$$\texttt{where} \begin{cases} (N_r, E_r) := \texttt{if}\ S(el_{e1}) = \emptyset\ \text{then}\ (\texttt{call}\ f)\ \texttt{else}\ \emptyset, f := \texttt{LkupMountingFunc}(el_{e_1}) \\ q_u := \{\texttt{if}\ (S(el_{e1}) \neq \emptyset \text{and}\ \texttt{Compare}(el))\ \text{then}\ \texttt{LkupUpdatingFunc}(el) \texttt{else}\ \emptyset\} \end{cases} \quad \text{(JSXElement)}$$

$$\frac{p \Rightarrow (N, E, el, q, S)}{(e, a, p) \Rightarrow (N, E, el, q, S)} \text{ (JSXClosingElement)} \quad \frac{p \Rightarrow (N, E, el, q, S)}{(e, a, p) \Rightarrow (N, E, el, q, S)} \text{ (JSXIdentifier)} \quad \frac{p \Rightarrow (N, E, el, q, S)}{(e, a, p) \Rightarrow (N, E, el, q, S)} \text{ (JSXElementName)}$$

Phase II: Updating (Async Events)

$$\frac{p \Rightarrow (N, E, el, q, S), (f, a.f, p) \Rightarrow (N_f, E_f, el, q, S)}{(\mathit{register}(x, f), a, p) \Rightarrow (N \cup N_f, E \cup E_f, el, q, S \cup \{a.x.name : o'\}), \forall o' \in \texttt{Child}^{a \to o}_{a.f}} \quad \text{(callback register)}$$

$$\frac{p \Rightarrow (N, E, el, q, S), (cb, a.cb, p) \Rightarrow (N_{cb}, E_{cb}, el, q, S), f := S(a.x.name), \texttt{call}\ f \Rightarrow (N_s, E_s, el, q, S)}{(\mathit{call}(x, cb), a, p) \Rightarrow (N \cup N_c b, E \cup E_c b, el, q \cup (\texttt{call}\ cb(o')), S), \forall o' \in \texttt{Child}^{a \to o}_{a.f}} \quad \text{(callback invocation)}$$

$$\frac{p \Rightarrow (N, E, el, q, S), (x, a.x, p) \Rightarrow (N_x, E_x, el, q, S)}{(\mathit{model}(x), a, p) \Rightarrow (N \cup N_x \cup \texttt{AddNode}^{o}_{a.x}, E \cup E_x, el, q, S)} \quad \text{(database model)}$$

$$\frac{p \Rightarrow (N, E, el, q, S), (e, a.e, p) \Rightarrow (N_e, E_e, el, q, S), (f, a.f, p) \Rightarrow (N_f, E_f, el, q, S), \texttt{if HasCommonKey}(\texttt{m}, \texttt{f}')\ \texttt{then Copy}(o', m) \Rightarrow (N_c, E_c)}{(x.\mathit{update}(f, e), a, p) \Rightarrow (N \cup N_e \cup N_c \cup N_f, E \cup E_e \cup E_c \cup E_f, el, q, S)}\ \texttt{where} \begin{cases} m := \texttt{Child}^{x}_{a \to o} \\ o' := \texttt{Child}^{a.e}_{a \to o} \\ f' := \texttt{Child}^{a.f}_{a \to o} \end{cases} \quad \text{(model update)}$$

$$\frac{p \Rightarrow (N, E, el, q, S), (e, a.e, p) \Rightarrow (N_e, E_e, el, q, S), m := \texttt{Child}^{x}_{a \to o}, n := \texttt{Child}^{a.e}_{a \to o}, \texttt{if HasCommonKey}(\texttt{m}, \texttt{n})\ \texttt{then Copy}(m, o) \Rightarrow (N_c, E_c)\ \texttt{where}\ o := \texttt{AddNode}^{a}_{o},}{(x.\mathit{find}(e), a, p) \Rightarrow (N \cup N_e \cup N_c, E \cup E_e \cup E_c, el, q, S)} \quad \text{(model read)}$$

Phase II: Updating (JSX Component Updating)

$$\frac{p \Rightarrow (N, E, el, q, S), (x, a.x, p, q) \Rightarrow (N_x, E_x, el, q, S)}{(\texttt{setState}(x), a, p) \Rightarrow (N \cup N_x, E \cup E_x \cup E_s, el, q \cup \{\texttt{if Compare}(el)\ \texttt{then LkupUpdatingFunc}(el) \texttt{else}\ \emptyset\}, S \cup S_x)}\ \texttt{where} \begin{cases} E_s := \texttt{AddEdge}^{v \to o}_{v_s \to o_s} \\ v_s := \texttt{LkupStateVar}(a.x) \\ o_s := \texttt{LkupObj}(a.x) \\ S_x :=< \texttt{LkupStateObjs}(el), \texttt{LkupPropsObjs}(el)\ >\} \end{cases} \quad \text{(setState)}$$

$$\frac{p \Rightarrow (N, E, el, q, S), (f, a.f, p) \Rightarrow (N_f, E_f, el, q, S), c := \texttt{LkupCleanupFunc}(el), (\mathit{call}\ c(), a.c, p) \Rightarrow (N_c, E_c, el, q, S), (e, a.e, p) \Rightarrow (N_e, E_e, el, q, S)}{(\texttt{useEffect}(f, e), a, p) \Rightarrow (N \cup N_f \cup N_e \cup N_c, E \cup E_f \cup E_e \cup E_c, el, q \cup \{N_d\}, S)\ \text{where}\ N_d := \texttt{Child}^{a \to o}_{a.f \to o'}} \quad \text{(useEffect)}$$

$$\frac{p \Rightarrow (N, E, el, q, S),}{(\texttt{forceUpdate}(), a, p) \Rightarrow (N, E, el, q \cup\ \{\texttt{LkupUpdatingFunc}(el))\}, S)} \text{ (forceUpdate)} \quad \frac{p \Rightarrow (N, E, el, q, S), (f(), a.f, p) \Rightarrow (N_f, E_f, el, q_f, S_f)}{(\mathit{call}\ f(), a, p) \Rightarrow (N \cup N_f, E \cup E_f, el, q \cup q_f, S \cup S_f)} \text{ (componentDidMount)}$$

$$\frac{p \Rightarrow (N, E, el, q, S), (f(\texttt{LkupPropsVar}(el), \texttt{LkupStateVar}(el)), a.f, p) \Rightarrow (N_f, E_f, el, q_f, S_f)}{(\mathit{call}\ f(a_1, \ldots, a_n), a, p) \Rightarrow (N \cup N_f, E \cup E_f, el, q \cup q_f, S \cup S_f)} \quad \text{(constructor, render, getDerivedStateFromProps, shouldComponentUpdate)}$$

$$\frac{p \Rightarrow (N, E, el, q, S), (f(S(el)), a.f, p) \Rightarrow (N_f, E_f, el, q_f, S_f)}{(\mathit{call}\ f(a_1, \ldots, a_n), a, p) \Rightarrow (N \cup N_f, E \cup E_f, el, q \cup q_f, S \cup S_f)} \quad \text{(getSnapshotBeforeUpdate, componentDidUpdate)}$$

Phase III: Unmounting

$$\frac{p \Rightarrow (N, E, el, q, S), (f(), a.f, p) \Rightarrow (N_f, E_f, el, q, S)}{((\mathit{call}\ f(), a, p) \Rightarrow (N \cup N_f, E \cup E_f, el, q, S)} \quad \text{(cleanup effects, componentWillUnmount)}$$

Figure 8: Detailed Operational Semantics for Building the Component Graph.