Remediating Superfluous Re-Rendering in React Applications

Anonymous Author(s)

ABSTRACT

React is an extremely popular framework for constructing user interfaces (UIs). A React UI is organized as a tree of components, each of which is defined by a function that returns a literal written in JSX, a syntactic extension of JavaScript consisting of a combination of XML tags, executable JavaScript code, and references to sub-components. React supports incremental re-rendering by maintaining an in-memory representation of a web page's Document Object Model (DOM) and automatically calculating a set of minimal changes that must be applied to the DOM when state changes occur.

However, React's semantics are complex and subtle, and programmers often write code that gives rise to unnecessary re-rendering, which hurts performance and responsiveness. We identify 5 React anti-patterns that give rise to unnecessary re-rendering, present a static analysis for detecting them, and rewrite rules that suggest how to refactor the code to improve rendering performance. The static analysis is potentially unsound, so developers should carefully review the suggested refactorings. A survey of 7,758 React repositories showed that 92.1% of them exhibit at least one anti-pattern, and careful experimental evaluation on 23 React projects revealed that the suggested refactoring reduces the number of rendering operations by 33.3% on average while preserving application behavior in all but one case. With a small increase in code complexity, we find an average reduction in rendering time of 20.54%, and three case studies reveal that the refactorings can greatly improve application responsiveness as the number of components scales.

1 INTRODUCTION

Modern-day web applications are complex pieces of software that aim to provide a rich user experience. One of the most common forms of web applications is a single-page application (SPA), where the content of the current page is updated dynamically instead of navigating to a new page. This approach enhances efficiency by eliminating the need to reload the foundational assets of the application—such as HTML, CSS, and JavaScript files—during navigation, thereby streamlining user interactions. This helps applications be *responsive*, one of the most desirable properties for user experience.

Building dynamic SPAs is nontrivial, and many UI libraries exist to help users develop them [1, 12, 17, 21, 23]. Typically, these libraries drastically change how developers use the language, e.g., through intricate APIs and syntactic extensions of the language. In this paper, we focus on React [21], which is the most widely used framework for creating SPAs in JavaScript. React is currently the most popular front-end UI framework for JavaScript and is used by popular websites such as Instagram, Netflix, GitHub, etc., and `w3techs.com` reports that, as of January 2025, 4.7% of all websites use React [31], including 22 *million* GitHub repositories [15]. It has also garnered the attention of the research community [6, 8, 9, 13, 19, 26]; e.g., Madsen et al. specified the formal semantics of React [19].

In React applications, developers define their UI in a *declarative* style. This enables the React engine to compute how much of the UI needs to be updated, or *re-rendered*, in response to user interaction. Unfortunately, the React engine is limited in its ability to detect situations where re-rendering can be avoided, and, if developers are not careful, their application may suffer from superfluous re-rendering, which may reduce responsiveness [9].

Our research aims to reduce the problem of unnecessary re-rendering in React applications. To this end, we carefully studied rendering behavior in a corpus of React applications and identified 5 anti-patterns that often give rise to needless re-rendering. We designed code transformations for each anti-pattern to reduce the number of rendering operations. We then developed a static analysis to detect instances of these anti-patterns and rewriting rules to automate code transformations necessary to eliminate them. Given the extreme dynamism of JavaScript, the static analysis is unsound and may potentially suggest transformations that change program behavior. Therefore, similar to recent work on refactoring for JavaScript [16, 28, 29], the code transformations should be viewed as *suggestions* that should be reviewed by a developer. The technique was implemented in a tool named *Reactor* that we plan to make available as open-source software in the near future.

An empirical evaluation of *Reactor* on 23 subject React applications revealed that the refactorings proposed by *Reactor* reduced the total number of rendering operations by 33.3% on average and did not introduce any behavioral differences in all but one transformed application. This reduction in renders results in 20.54% less time spent rendering on average, and an additional set of experiments on three applications with a scalable number of components shows appreciable improvements in application responsiveness as the number of components increases. The refactorings also only contribute to a modest increase in code complexity. In another experiment, the anti-pattern detection queries were run on a corpus of 7,758 React repositories. This experiment identified instances of the anti-patterns in 92.1% repositories, providing evidence that these anti-patterns are prevalent beyond the subject applications studied in the empirical evaluation.

In summary, the contributions of this paper are:

(1) The identification of a set of ***anti-patterns*** that often give rise to unnecessary re-rendering in React applications,
(2) A ***static analysis*** for detecting anti-pattern instances and ***rewriting rules*** describing remedial code transformations,
(3) An ***implementation*** of the static analysis and rewriting rules in an automated tool called *Reactor*, which we plan to release as open-source software, and
(4) An ***empirical evaluation*** of *Reactor* on 23 subject applications, demonstrating that it reduces the total number of rendering operations in most cases.

This paper is organized as follows. Section 2 provides background on React. Section 3 presents a motivating example that exhibits superfluous re-rendering and its remediation. Section 4 describes the identified anti-patterns. Section 5 describes our static analysis and code transformation techniques. Section 6 presents our evaluation. We discuss potential limitations in Section 7 and related work in Section 8. We conclude with a summary of our findings.

2 BACKGROUND

React [21] is a powerful JavaScript library for building interactive User Interfaces (UIs) that has been embraced enthusiastically by the web development community.

Components. A React application comprises a collection of ***components*** corresponding to elements of an application's UI. By structuring UIs in this way, developers can build complex UIs from simple, modular pieces, facilitating reuse. React components can be functions or classes. We only consider functional components, which is currently the preferred mechanism.

React syntax. React applications are typically written in JSX[1], a syntactic extension of JavaScript in which XML values may occur as literals[2]. JSX literals can be turned into ***template literals*** that contain ***placeholders*** consisting of a JavaScript expression surrounded by curly braces. At run time, such embedded expressions are executed to obtain concrete values with which the template is instantiated, resulting in concrete HTML values that a browser can render. An embedded expression that evaluates to an array is turned into a sequence of HTML elements. Syntactically, functional React components are simply JavaScript functions that return a JSX literal. Figure 1(a) shows an example React application consisting of two components, `CounterButton` and `Counter`. The `CounterButton` component consists of a header containing a text message followed by a simple button, which is expressed by the JSX literal on lines Lines 2-3. A React component may include another React component by simply referring to the latter's name between angular brackets, called a ***JSX tag***. In the example, the `Counter` component defines a UI element that contains a text form in which users can enter their name, followed by a `CounterButton`'s JSX tag (lines 11-19).

State and Props. React components may have ***state***, i.e., values that persist until explicitly changed or the component is destroyed. State elements are declared by invoking React's ***useState*** function, which takes the initial value as an input and returns the state and a ***state-setter*** function that must be used to change the state. E.g., the `Counter` component maintains `counter` and `name` as its state. On lines 5-6, these are initialized to the values `0` and `'Guest'`, and on lines 8, and 9 the `setCount`, and `setName` state-setters are invoked.

Components may pass values—commonly referred to as ***properties*** or ***props*** in React parlance—to their sub-components (also known as their *child components*) by including them as key-value pairs when referencing the sub-component. For example, on lines 16–18, the `Counter` component passes values to `CounterButton` bound to its props `message` and `onClick`, so `CounterButton` is a child of `Counter`.

Rendering. Rendering a React component involves executing its code to produce HTML. A component re-renders if its state, or one of its ancestor's state, changes. One of the key strengths of React is its support for incremental re-rendering of an application's UI. The *virtual DOM* (VDOM) is central to this efficiency. The VDOM is a simplified, in-memory representation of the actual Document Object Model (DOM). Unlike the DOM, a complex, live structure rendered by the browser, the VDOM is a lightweight abstraction that allows React to perform updates efficiently. When a state change occurs, React creates a new version of the VDOM. This new version is then compared to the last snapshot of the VDOM, a process known as ***diffing***. During diffing, React identifies the exact differences between the old and new VDOMs and computes the minimal set of changes needed to update the actual DOM in a process referred to as ***reconciliation***. This selective update process ensures the browser re-renders only the changed parts of the DOM rather than the entire DOM. This targeted updating enhances performance significantly by reducing the amount of DOM manipulation required, as browser re-rendering is time-consuming.

State Management. React applications adopt a centralized state management approach where the nearest common ancestor component maintains the state needed by several sub-components. This ancestor makes the state accessible to its children, passing it as props in their JSX tags. This may include event handlers, allowing child components to update the state also accessed by its parent and/or sibling sub-components. Consequently, any change in the ancestor's state prompts re-rendering of the ancestor and its descendants, ensuring UI consistency across the component hierarchy.

This is demonstrated in the example snippet, where `Counter` component passes `handleClick`, a function that uses `setCount` to update its state, as the `onClick` prop to its child (Figure 1(a), line 18). When the button is clicked, `onClick` handler of the `<button>` element invokes the `handleClick` function from the `Counter` component. Inside `handleClick`, the `setCount` state-setter is called, causing the `Counter` component to be re-rendered. As a descendant, the `CounterButton` component also re-renders, ensuring the UI reflects the updated state.

Hooks. React's diffing and reconciliation mechanisms aim to ensure efficient updates to the DOM. However, as we shall see in Section 4, React may occasionally re-render components unnecessarily, i.e., without causing changes in the DOM. Fortunately, React provides a number of mechanisms for functional components that provide more fine-grained control over the rendering process. These are known as ***hooks*** because they enable one to "hook into" React's state and life-cycle management. The `useState` function we saw previously is an example of React hooks and provides access to a component's state. We now discuss a number of other hooks.

- ***useRef*** enables value persistence across rendering operations without triggering a re-render when the stored mutable value is updated. This hook is commonly used to keep a reference to a DOM element or to keep track of mutable data whose change does not require a re-render of the component. It returns a reference object with a *.current* property where the stored value can be accessed or modified.
- ***Memoization*** is a performance optimization technique for preventing unnecessary re-rendering by performing a shallow comparison between the current and new props and only re-rendering the component if there is a difference. A functional component can be memoized by wrapping it in ***React.memo***[3]. React maintains a copy of the VDOM for comparison, so memoization does not incur a significant overhead.

[1]JSX is an acronym for "JavaScript XML".
[2]React applications can also use TSX, an equivalent extension of the TypeScript.

[3]React.memo is a higher-order component that enhances a functional component by wrapping it. Wrapping in this context means that React.memo takes the original component as an argument and returns a new component with added functionality.

```
function CounterButton({ message, onClick }) {
  return (<div><h1>{message}</h1>
              <button onClick={onClick}>Click me!</button> </div>);}
function Counter() {
  const [count, setCount] = useState(0);
  const [name, setName] = useState('Guest');
  const handleClick = () => {
    setCount(count + 1); };
  const handleNameChange = (e) => { setName(e.target.value);  };
  return (
    <div>
      <input type="text"
        value={name}
        onChange={handleNameChange}
        placeholder="Enter your name" />
      <CounterButton
        message={`Hi ${name}, you clicked ${count} times.`}
        onClick={handleClick} />
    </div>);}
```

(a)

```
1  const CounterButton=React.memo(({message, onClick})=>{
2    return (<div><h1>{message}</h1>
3            <button onClick={onClick}>Click me!</button></div>);}
4  function Counter() {
5    const [count, setCount] = useState(0);
6    const nameRef = useRef('Guest');
7    const handleClick = useCallback(() => {
8      setCount((prevCount) => prevCount + 1);}, []);
9
10   return (
11     <div>
12       <input type="text"
13         ref={nameRef}
14
15         placeholder="Enter your name" />
16       <CounterButton
17         message={`Hi ${nameRef.current.value}, you clicked ${count} times.`}
18         onClick={handleClick} />
19     </div>);}
```

(b)

Figure 1: (a) Example React application. (b) Optimized version where components are re-rendered less often.

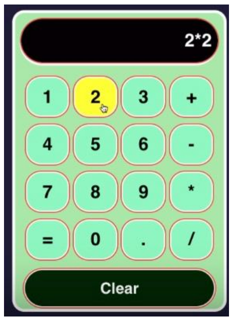
- React's ***useCallback*** hook memoizes functions within components: it returns a version of a callback function that only updates when its dependencies (specific states or props that, if altered, necessitate an update) change. This is useful when passing callbacks to child components, specifically those wrapped in `React.memo`, which compares props by reference. By keeping the function reference unchanged across re-renders, `useCallback` helps prevent child components from redundant re-rendering.

Figure 1(b) shows a variation of the example of 1(a) that uses these `useRef`, `useCallback`, and `React.memo` functions. On line 6, `useRef` is used to initialize a variable `nameRef` to "Guest". This reference tracks the user's name from the input element (line 13), updating as the input value changes. These updates do not trigger a component re-render on every key press. Instead, the updated value becomes visible only after the `Counter` component re-renders, which occurs upon a button click. Figure 1(b) also illustrates memoization: the child `CounterButton` is wrapped in `React.memo` (line 1), signaling React to bypass re-rendering it if the current and new props are the same. Also, the `handleClick` function is wrapped in `useCallback` (line 7) to ensure that the reference to the `onClick` prop remains the same when its parent component `Counter` re-renders.

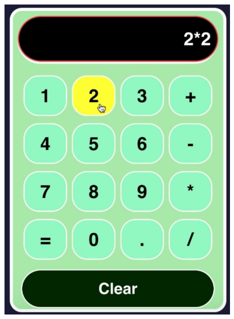
Having these memoization steps applied, the `CounterButton` component will not re-render if `Counter` updates without altering `message` or `handleClick`. This example highlights the two different ways of handling form data in React. Figure 1(a) presents an example of a ***controlled component***, where the React component's state handles form data. On the other hand, Figure 1(b) presents an instance of an ***uncontrolled component*** where the DOM handles form data.

3 MOTIVATING EXAMPLE

Figure 2(a) shows a screenshot of **Calculadora** [2], a basic open-source calculator built using React in which needless re-rendering occurs. We developed a profiling tool that highlights re-rendered components by drawing a red box around them. The screenshot shows the UI while running it with our re-rendering profiler after the user has entered a formula `2*2`. In Figure 2(a), each button is surrounded by a red box, indicating that it was re-rendered. Intuitively, one would expect only the display to be re-rendered when a button is pressed and the buttons' UI could remain unchanged.



(a) Before transformation



(b) After transformation

Figure 2: Re-render comparison before vs. after memoization when a button is clicked, red indicating components that were re-rendered.

To understand why the buttons re-render, we review the relevant code in Figure 3(a). Each button is represented by a `Button` component, and the calculator's display is modeled by a `Screen` component, all of which are children of a top-level `App` component. `App` maintains an `input` state, representing `Screen`'s content, used for performing calculations. On lines 4-5, function `addInput` is declared that concatenates a value to the `input`. This function is passed as the `handleClick` prop to each `Button`. Hence, clicking a button will invoke `addInput`, causing `App`'s state-setter `setInput` to be invoked. As we discussed in Section 2, *any* update to the `App` component's state triggers re-rendering of *all* its child components.

Inspecting the code reveals that `Button` does not have mutable state of its own and that the same function, `addInput`, is always passed as a property. In such cases, React's memoization mechanism can be employed to prevent unnecessary re-rendering. This requires the two transformations shown in Figure 3(b):

- The `Button` component is memoized using `React.memo` (line 11). This instructs React not to re-render it when its parent `App` does, *provided that the same props are passed.*
- The `addInput` function passed as a prop from `App` to `Button` is memoized using `useCallback` (line 4). Without memoization, `App` recreates the `addInput` function on each re-render, thereby creating a new reference for the same function. Wrapping the function in `useCallback` and passing in an empty dependency list circumvents this, thus avoiding unnecessary re-renders.

```
import { useState } from 'react';
function App() {
  const [input, setInput] = useState('');
  const addInput = val => {
    setInput(input + val); };
// ...
  return ( /* ... */
    <Screen input={input}/>
    <Button handleClick={addInput}>1</Button>
    /* ... */ );}
function Button(props) { /* ... */ }
```

(a)

```
import { useState, useCallback } from 'react';
function App() {
  const [input, setInput] = useState('');
  const addInput = useCallback((val) => {
    setInput((currentInput) => currentInput + val); }, []);
  // ...
  return ( /* ... */
      <Screen input={input}/>
      <Button handleClick={addInput}>1</Button>
      /* ... */ );}
const Button = React.memo(function Button(props) { /* ... */ }
```

(b)

Figure 3: Example: unnecessary re-rendering in Calculadora. (a) Non-memoized. (b) Memoized.

We tested the optimized code and confirmed that extra re-renders were eliminated while preserving application behavior, evidenced by the absence of red borders around buttons in Figure 2(b). Eliminating these re-renders results in 31.3% less time spent on average rendering the application when a button is pressed.

4 ANTI-PATTERNS

The example laid out in the last section is an example of a broader *anti-pattern* that leads to re-rendering, and this section describes our methodology for identifying other such common anti-patterns that give rise to needless rerendering in React applications.

4.1 Anti-pattern detection Methodology

To identify anti-patterns, we randomly sampled 10,000 GitHub repositories that listed React as a dependency. We attempted to build the CodeQL [14] databases for these repositories, which succeeded for 7,758 repositories. Repositories for which we could not build a CodeQL database were excluded from further analysis, as a database is required to run our analysis queries. We wrote a static analysis using CodeQL to identify React components where at least one child component did not depend on all of its parent's state, representing likely scenarios with needless re-rendering. We ran this static analysis on the 7,758 repositories, which identified 2,047 repositories for further analysis.

To construct a set of repositories for manual investigation, we iteratively sampled from the 2,047 candidates and applied a series of filtering criteria. A repository was retained only if it (1) was still available on GitHub, (2) could be built and installed locally, (3) could be executed and interacted with, (4) exhibited visual indications of unnecessary re-rendering (as indicated by the profiler tool) despite no visible change in its appearance, and (5) did not require significant structural changes due to excessive code complexity or deep reliance on external libraries. This process continued until we observed the same anti-patterns emerging repeatedly, at which point we considered the search saturated.

In cases where remediating re-rendering did not require such significant changes, we studied and attempted to remediate the observed re-rendering manually. This filtering process yielded 40 repositories suitable for detailed analysis, of which we found and fixed unnecessary re-rendering issues in 14.

4.2 Anti-Pattern 1: *Controlled Component*

Figure 1(a) shows an example of a controlled component (`Counter`) declaring a state variable `name` and an associated setter `setName` (line 6). In the JSX returned by this component, `name` is referenced on line 13 and `setName` is invoked through `handleNameChange` on line 14. Here, each update to the input element will cause the component to re-render because one of its state-setters is called. E.g., in Figure 1(a), each keystroke triggers a re-render, which is unnecessary.

To avoid unnecessary re-rendering in this type of scenario, one can rewrite the application so that the DOM handles the form data. This approach, commonly referred to as an *uncontrolled component*, is illustrated in Figure 1(b), and involves using React's `useRef` hook to create a reference object with a "`.current`" property, allowing values to persist across re-renders without triggering new re-renders when modified. Now, changing the controlled component of Figure 1(a) into an uncontrolled component of Figure 1(b) involves the following changes: (i) importing the `useRef` hook, (ii) creating a reference `nameRef` for the `input` element (line 6), (iii) extending the form with a reference `ref` attribute bound to the reference `nameRef` (line 13), and (iv) getting/setting the value of the `input` element by accessing it via the reference object (line 17).

4.3 Anti-Pattern 2: *State Var. not Affecting UI*

Developers may use state variables for internal logic rather than updates to the UI, which may cause components to re-render unnecessarily. Figure 4(a) shows a component `CompFuncCicloVida`, taken from one of the 14 repositories under study (*16-feb-react-grupo-1*). Here, the `miVariable` state variable does not impact the UI, but it is updated by a state-setter in the `updateVar` function (line 6), which is bound to the button's `onClick` handler (line 9), the `CompFuncCicloVida` component is re-rendered after each button click.

In such cases, the problem can be remediated by using React's `useRef` hook to introduce a reference object so that changes to the variable's value no longer cause re-rendering. Figure 4(b) shows how the code of Figure 4(a) can be transformed to prevent the unnecessary re-rendering by: (i) importing the `useRef` hook (line 1), (ii) replacing the declaration of the target state variable with a reference object (line 3), and (iii) replacing any access to (or modification of) that state variable with access to (or modification of) the "`.current`" property of the reference variable (lines 6, 7, 8).

4.4 Anti-Pattern 3: *Propless Child Component*

As outlined in Section 2, re-rendering a React component causes all its descendent components to re-render to guarantee UI consistency. In cases where components do not receive props, they inherently exhibit no change in their rendered output upon parent component re-rendering. Memoizing a prop-less component means that the shallow comparison of props trivially succeeds (given that there are none), enabling React to bypass the re-rendering process.

```
import { useState, useEffect } from "react"
const CompFuncCicloVida = () => {
    const [miVariable, setMiVariable] = useState<boolean | undefined>()
    // ...
    const updateVar = () => {
       setMiVariable((val) => {
            if (typeof val === "undefined") return true
            return !val });}
    return (<div><button onClick={updateVar}> Actualiza </button></div>)}
```

(a)

```
import {useState,useEffect,useRef} from "react"
const CompFuncCicloVida = () => {
    const miVariableRef = useRef<boolean | undefined>()
    // ...
    const updateVar = () => {
       miVariableRef.current =
        typeof miVariableRef.current==="undefined"
                ? true : !miVariableRef.current }
    return (<div><button onClick={updateVar}> Actualiza </button></div>)}
```

(b)

Figure 4: pattern *State Variable not Affecting UI* example: (a) Uses state. (b) Uses reference.

4.5 Anti-Pattern 4: *Child Component with Object or Array Element Prop*

Figure 5(a) shows a Tic-Tac-Toe game implementation exhibiting this anti-pattern. Here, the `App` component defines the game's grid structure, with a `Square` component for each grid square which receives a `value` prop that is either "X", "O", or empty. Each `Square`'s `value` comes from the `squares` array (a state variable in `App`) and changes only once, even though `squares` is updated with each action.

Each time a player clicks a `Square` in the game grid, the `onClick` handler, bound to the `handleClick` prop received from the `App` component, is called, which calls the `setSquares` state-setter in turn and triggers `App` to re-render. Hence, in accordance with React's strategy of re-rendering all child components when a parent component re-renders, all `Squares` re-render when their parent `App` re-renders.

Figure 5(b) shows how re-rendering can be prevented. Here, `Square` is wrapped in `React.memo` (line 1), signaling React to bypass re-rendering of `Square` components with unchanged props. Since each `Square`'s `value` prop only changes when it is first clicked, this prevents re-rendering of the non-clicked squares as long as the value passed to `handleClick` also remains unchanged.

4.6 Anti-Pattern 5: *Child Component with Function Prop*

Figure 3 shows an example where a parent component `App` passes the `addInput` function as `handleClick` prop to its child component `Button` (line 9). When a button is clicked, the `handleClick` handler invokes `addInput`, which in turn calls `App`'s `setInput`, triggering a re-render of `App`. However, functions defined in a component are ***re-defined*** each time it is rendered. Consequently, all buttons re-render, even if `Button` is memoized, because `addInput` is recreated on each re-render of `App`, passing new references to the `Button` instances.

In such cases, wrapping a function in `useCallback` ensures that its identity remains unchanged across re-renders unless its dependencies, specified in the dependency array, change. As a result, utilizing the `useCallback` React hook can prevent these redundant re-renders by preserving the reference of the handler function across re-renders. Figure 3(b) shows how the code can be rewritten using a memoized handler(line 4) to avoid the unnecessary re-rendering. In particular, by memoizing both the `Button` component (line 11) and its `handleClick` prop ensures that only the clicked button re-renders, preventing unnecessary re-renders of the other buttons.

5 DETECTING AND REMEDIATING ANTI-PATTERNS

The previous section used code examples to illustrate a number of anti-patterns that give rise to needless re-rendering along with the code changes required to remediate them. This section defines the anti-patterns and changes more formally through a set of declarative rewrite rules. Our tool (*Reactor*) implements the static analyses and code transformations required to realize these rewrite rules.

5.1 Rewrite Rule Syntax

The general format for a rewrite rule is as follows:

$$\frac{\textit{preconditions}}{\textit{old code} \;\longrightarrow\; \textit{new code}} \qquad (\textsc{Rule-Name})$$

The premise of the rule includes *preconditions* that must be met in order for the transformation to take place. These preconditions represent some facts obtained using static data flow analysis or analysis of the AST. In formulating the rules, we will rely on auxiliary functions such as **replaceExpr**(A, e, e'), which takes some AST node A and replaces occurrences of expressions e with e'. We will also introduce similar auxiliary functions such as **replaceCalls** for call sites, **replaceDecl** for declarations, and so on.

5.2 *Controlled Component*

Rule Controlled-Component is concerned with transforming controlled components into uncontrolled components and applies to a set of import statements I and a React component `F` with a body B, denoted by I ... `function F (...)` B. Reading the rule from top to bottom, the rule states that the body B of the component contains a declaration D of a state variable `v`, with associated state setter $\texttt{set}_\texttt{v}$, that is initialized to `e`. Moreover, F's return expression contains a JSX fragment R_j corresponding to either a Form or Input component, where a function f is passed as a change handler. Notably, f calls the state setter $\texttt{set}_\texttt{v}$. As is the case with all subsequent rules, these facts are all computed using static analysis.

Then, a new declaration D' is defined for a reference variable $\texttt{v}_\texttt{ref}$ (also initialized to `e`); a new JSX fragment R'_j is obtained from R_j by replacing the `onChange` attribute with a `ref` attribute that refers to $\texttt{v}_\texttt{ref}$. The body is then modified several times, in order: B' from B where the new fragment R'_j replaces R_j, B'' from B' where the new reference variable declaration D' replaces the old state variable declaration D, B''' from B'' where references to the state variable `v` are replaced with accesses to the "current.value" property of the new reference variable $\texttt{v}_\texttt{ref}$, and finally B'''' from B''' where calls to the state setter $\texttt{set}_\texttt{v}$ are replaced with assignments to the new reference variable $\texttt{v}_\texttt{ref}$. The set of imports is expanded to include an import to `'useRef'`, and thus the transformed code is given by I' ... `function F (...)` B''''.

```
function Square({handleClick, value}) { /* ... */ }
function App() {
    const [squares, setSquares] = useState(Array(9).fill(null));
    // ...
    return ( /* ... */
      <Square value={squares[0]} handleClick={() => handleClick(0)} />
      /* ... */
    );}
```

(a)

```
const Square = React.memo(({handleClick, value}) => { /* ... */ }
function App() {
    const [squares, setSquares] = useState(Array(9).fill(null));
    // ...
    return ( /* ... */
      <Square value={squares[0]} handleClick={() => handleClick(0)} />
      /* ... */
    );}
```

(b)

Figure 5: pattern *Child Component with Object or Array Element Prop* example: (a) Non-memoized. (b) Memoized.

$$\frac{\begin{array}{c} B \text{ contains declaration } D = \texttt{const [v, set}_\texttt{v}\texttt{] = useState(e)} \\ \textbf{returnExpr}(B) \text{ contains } R_j = \texttt{<}N \ldots,\ \texttt{onChange} = f, \ldots\texttt{>} \\ N \text{ is a Form or Input} \qquad f \text{ calls } set_v \\ D' = \texttt{const v}_\texttt{ref} = \texttt{useRef(e)} \\ R'_j = \textbf{replaceAttr}(R_j, \texttt{onChange} = f, \texttt{ref} = \texttt{v}_\texttt{ref}) \\ B' = \textbf{replaceExpr}(B, R_j, R'_j) \qquad B'' = \textbf{replaceDecl}(B', D, D') \\ B''' = \textbf{replaceVarRefs}(B'', \texttt{v}, \texttt{v}_\texttt{ref}\texttt{.current.value}) \\ B'''' = \textbf{replaceCalls}(B''', \texttt{set}_\texttt{v}\texttt{(e)}, \texttt{v}_\texttt{ref} = \texttt{e}) \\ I' = I \cup \{\texttt{import \{useRef\} from 'react'}\} \end{array}}{I \ldots \texttt{function F} (\ldots)\ B \rightarrow I' \ldots \texttt{function F} (\ldots)\ B''''} \ (\textsc{Controlled-Component})$$

$$\frac{\begin{array}{c} B \text{ contains declaration } D = \texttt{const [v, set}_\texttt{v}\texttt{] = useState(e)} \\ \text{no data flow from } \texttt{v} \text{ to } \textbf{returnExpr}(B) \\ D' = \texttt{const v}_\texttt{ref} = \texttt{useRef(e)} \\ B' = \textbf{replaceDecl}(B, D, D') \\ B'' = \textbf{replaceVarRefs}(B', \texttt{v}, \texttt{v}_\texttt{ref}\texttt{.current}) \\ B''' = \textbf{replaceCalls}(B'', \texttt{set}_\texttt{v}\texttt{(e)}, \texttt{v}_\texttt{ref} = \texttt{e}) \\ I' = I \cup \{\texttt{import \{useRef\} from 'react'}\} \end{array}}{I \ldots \texttt{function F} (\ldots)\ B \rightarrow I' \ldots \texttt{function F} (\ldots)\ B''''} \ (\textsc{State-Not-Affecting-UI})$$

$$\frac{\begin{array}{c} \texttt{F} \text{ is a React component} \\ I' = I \cup \{\texttt{import \{memo\} from 'react'}\} \end{array}}{I \ldots \texttt{function F () } B \rightarrow I' \ldots \texttt{const F = memo(function() } B\texttt{)}} \ (\textsc{Propless-Component})$$

$$\frac{\begin{array}{c} \texttt{F} \text{ is a React component} \\ \exists \texttt{<}F \ldots, p = e, \ldots\texttt{>} \mid \texttt{e = v.f} \text{ or } \texttt{e = v[i]} \text{ for some state variable } \texttt{v} \\ I' = I \cup \{\texttt{import \{memo\} from 'react'}\} \end{array}}{I \ldots \texttt{function F} (\ldots)\ B \rightarrow I' \ldots \texttt{const F = memo(function(}\ldots\texttt{) } B\texttt{)}} \ (\textsc{Memoize-Component-Indexes-State})$$

$$\frac{\begin{array}{c} B \text{ contains function declaration } D = \texttt{function f} (\ldots)\ \texttt{B}_\texttt{f} \\ B \text{ contains component definition } C = \texttt{function N} (\ldots)\ \texttt{B}_\texttt{N} \\ \textbf{returnExpr}(B) \text{ contains } \texttt{<}N \ldots,\ \texttt{attr} = f, \ldots\texttt{>} \\ L = \text{all state variables referenced in } \texttt{B}_\texttt{f} \\ D' = \texttt{const f = useCallback(function} (\ldots)\ \texttt{B}_\texttt{f}\texttt{, L)} \\ C' = \texttt{const N = memo(function} (\ldots)\ \texttt{B}_\texttt{N}\texttt{)} \\ B' = \textbf{replaceDecl}(B, D, D') \qquad B'' = \textbf{replaceDecl}(B', C, C') \\ I' = I \cup \{\texttt{import \{useCallback, memo\} from 'react'}\} \end{array}}{I \ldots \texttt{function F} (\ldots)\ B \rightarrow I' \ldots \texttt{function F} (\ldots)\ B''} \ (\textsc{Function-Passed-As-Prop})$$

$$\frac{\begin{array}{c} B \text{ contains declaration } D = \texttt{const [v, set}_\texttt{v}\texttt{] = useState(e)} \\ B \text{ contains component definition } C = \texttt{function N} (\ldots)\ \texttt{B}_\texttt{N} \\ \textbf{returnExpr}(B) \text{ contains } \texttt{<}N \ldots,\ \texttt{attr} = \texttt{set}_\texttt{v}, \ldots\texttt{>} \\ C' = \texttt{const N = memo(function} (\ldots)\ \texttt{B}_\texttt{N}\texttt{)} \\ B' = \textbf{replaceDecl}(B, C, C') \\ I' = I \cup \{\texttt{import memo from 'react'}\} \end{array}}{I \ldots \texttt{function F} (\ldots)\ B \rightarrow I' \ldots \texttt{function F} (\ldots)\ B'} \ (\textsc{Setter-Passed-As-Prop})$$

Figure 6: Rewrite rules describing the remedial code transformations.

5.3 *State Variable not Affecting UI*

Many of the code transformation steps for this rule are shared with the previously discussed rewrite rule, and we will not discuss them in detail here. The primary difference lies in the second clause: the code should be transformed if there is no data flow between a state variable v and the return expression of component `F` (i.e., the state variable is not passed as a prop to any returned component).

5.4 *Propless Child Component*

Rule Propless-Component describes the memoization of propless React components. Here, `F` is a React component, and the (function) component declaration is transformed into an assignment wrapping the function in a call to `memo` (as `memo` cannot be directly called on function declarations), and as before the imports must be extended to import `memo` from `'react'`.

5.5 *Child Component with Object or Array Element Prop*

Rule Memoize-Component-Indexes-State applies to situations where one of a component's props receives data from a state variable through property access or array access to that variable. Note that `F` is a React component. If there exists an instantiation $<F\ldots, p = e, \ldots>$ of this component anywhere in the project where some prop p is passed either a property access `v.f` or index into an array `v[i]` of some state variable v, then the component should be memoized. As in the previous rewrite rule, the component is memoized by transforming the function declaration into an assignment of a function literal, where the function creation is wrapped in `memo`. As before, the imports I are updated to import `memo`.

5.6 *Child Component with Function Prop*

For the last anti-pattern, we distinguish two cases: (a) a locally declared function is passed as a prop to a child component, and (b) a setter function is passed as a prop to a child component.

Rule Function-Passed-As-Prop applies when a function is passed as a prop to a component. Here, function body B contains both a function declaration D for a function `f` and a React component declaration C for a component `N`. The return expression of B contains a JSX expression that instantiates the component `N`, and notably, that instantiation receives `f` as a prop. To remediate this anti-pattern, first, the list L of "dependencies" of `f` is obtained, consisting of all state variables that are referenced in the body of `f`. Then, D' is a new declaration that wraps the creation of `f` in `useCallback`, and this call to `useCallback` also takes L as an argument. Moreover, C' is a declaration of `N` that also memoizes `N`. Two transformations are then applied: B' is obtained from B where the function declaration D is replaced with D', and B'' from B' where the component declaration C is replaced with C'. The imports I are extended with imports to `useCallback` and `memo`, giving I', and the final code is given by I' … `function F` (…) B''. (Note that the transformation can span multiple scopes or even files, and this simplified version is shown for illustrative purposes.)

Finally, rule SETTER-PASSED-AS-PROP applies when a function passed to a component is a state setter. Remediating this anti-pattern is similar to the above case, except that the setter does not need to be wrapped in `useCallback`. In body B, a state variable is declared (denoted D, with state setter $\texttt{set}_\texttt{v}$) and a component N is declared (C). Here, the component is instantiated with $\texttt{set}_\texttt{v}$ passed as a prop (see $<N$ ..., $\texttt{attr} = \texttt{set}_\texttt{v}$, ...>). To remediate the pattern, the declaration of component N is wrapped in a call to memo (C'), and a new body B' is obtained from the old B by replacing the declaration C with C'. I' is obtained from I by adding an import to memo, and the transformed code is given by I' ... function F (...) B'.

5.7 Implementation

These rewrite rules were implemented in a tool called *Reactor*. The implementation leverages the CodeQL [14] static analysis framework to detect the anti-patterns and to collect the data flow information required for the remedial program transformations, which are implemented using JavaScript's Babel parsing and AST infrastructure. More precisely, CodeQL queries compute all of the preconditions of a rule, and the JavaScript rewrite tool takes this information to transform the code. ***Our tool is available at this Zenodo link.***

6 EVALUATION

This evaluation aims to answer the following research questions:

- **RQ1** How prevalent are the anti-patterns?
- **RQ2** How often do the automated transformations introduce behavioral differences?
- **RQ3** How do the transformations impact the number of re-rendering operations?
- **RQ4** What is the impact of the transformations on performance?
- **RQ5** What is the impact of the transformations on code complexity?
- **RQ6** What is the cost of the analysis?

In this Section, we start by depicting the experimental methodology, clarifying the subject selection process for our analysis in Section 6.1. We then attempt to answer each of the research questions with the dedicated corpus for analysis in the following subsections.

6.1 Experimental Methodology

To answer RQ1, we ran the anti-pattern identification queries we developed on all 7,758 repositories. RQs 2, 3, 4, and 5 however, cannot be answered "at scale" like RQ1. Although applying the analysis and transformations requires no manual effort, confirming that no behavioral differences were introduced and checking if the number of needless re-renders decreased is a manual effort. Therefore, we attempted to select a subset of that corpus whose results could be representative of the overall results.

Our evaluation set began with the 14 repositories described in Section 4, which we had previously analyzed and manually fixed. These projects provided a basis for evaluating the effect of *Reactor* on repositories that we had already verified to be fixable manually. To assess generalizability, we then augmented this set with repositories that had not been seen during manual exploration.

We returned to the full list of 10,000 repositories and identified 75 that had published at least one release containing a downloadable asset, indicating some level of real-world usage. [4]

Reactor detects at least one anti-pattern in 57 of these projects. From these, we randomly sampled repositories until we found 9 (that did not use class components) that we could install, build, set up, and run locally. We used the Chrome Dev Tools React toolkit (the Components and Profiler tabs) and our own React Dynamic Profiler to observe and understand an application's rendering behavior.

In total, we evaluated 23 repositories: the original 14 manually analyzed projects, and an additional 9 previously unseen repositories selected through the process above.

We took the following steps to answer each research question:

- For **RQ1**, we developed CodeQL [14] queries to identify each of the anti-patterns specified in Section 5. This large-scale analysis was done on the corpus of 7,758 repositories. (These CodeQL queries are available in the appendix.)
- **RQ2** and **RQ3** require non-trivial manual effort; **RQ2** requires manual inspection of projects pre- and post-refactoring. For **RQ3**, we instrumented the code to count how often a component renders before and after refactoring. Since many React projects lack tests, we created one "scenario" per studied application interacting with at least one component exhibiting an anti-pattern. We then carefully compared the application's behavior and the number of renders before and after refactoring for 23 subject repositories.
- For **RQ4**, we used the Chrome Dev Tools React Profiler tab to compute the amount of time taken to render when following the aforementioned scenarios for the 23 subject applications. In addition, we collected three additional applications with a scalable number of components to investigate the impact of re-rendering on performance as applications scale.
- For **RQ5**, we used escomplex [3] to measure different code complexity metrics such as SLOC, Cyclomatic complexity, and Halstead Complexity, both before and after transformation for the 23 subject repositories.
- Finally, for **RQ6**, we measured the time to run *Reactor* on the smaller set of 23 applications using the Unix `time` utility. We also timed the most time-consuming step of the analysis in a larger-scale study over the full set of 7,758 databases.

All the CodeQL queries were run, using npm version v9.6.6, Node.js version v18.16.0, and CodeQL version v2.17.2.

6.2 RQ1: How prevalent are the anti-patterns?

We ran the anti-pattern detection queries on the corpus of 7,758 projects, and we found at least one instance of an anti-pattern in 92.1% of projects. The anti-patterns appeared as follows: *Controlled Component* in 30.4% of applications, *State Variable not Affecting UI* in 42.7%, *Propless Child Component* in 91.2%, *Child Component with Object or Array Element Prop* in 6.3%, and *Child Component with Function Prop* in 36.7%. These results indicate that the anti-patterns occur frequently in real-world React applications.

[4]A project needs at least one release (made up of one or more assets, e.g., a zip file of source code) with at least one downloaded asset. This is quite restrictive, as it indicates the project has at least one official release that has garnered some attention; in our experience, the vast majority of React projects on GitHub do not meet this standard.

Table 1: Overview of results. The first row of the table reads: in 16-feb-react-grupo-1, 0 instances of anti-pattern P1, 1 of P2, 11 of P3, 0 of P4, and 1 of P5 were detected. The number of component renders in the scenario we studied was reduced by 16% after refactoring. Rendering operations took on average 4.83ms before, and 4.45ms after refactoring; an improvement of 7.9%. It took 6s to build the CodeQL database, and 688s to run all queries for this application. The cyclomatic complexity changed from 325 to 338 after refactoring. Bold numbers in % Time Saved indicate a statistically significant difference between performance before and after refactoring. We omit render reductions and time spend rendering for TwitchKillMe as behavioral differences were observed in this application.

	Anti-pattern Detected					Render	Avg. Time Rendering (ms)		% Time	Cyclomatic		Analysis time (s)	
Subject Repository Name	**P1**	**P2**	**P3**	**P4**	**P5**	**Red. %**	**Before**	**After**	**Saved**	**Before**	**After**	**DB Build**	**Query**
16-feb-react-grupo-1	0	1	11	0	1	16%	4.83	4.45	7.9%	325	338	6	688
26-react-star-wars-hooks	0	0	8	0	1	20%	7.72	7.43	3.8%	205	271	5	852
30days-30projects	9	12	38	1	1	50%	51.61	27.87	**46%**	1107	1131	7	852
age-react	0	0	3	0	2	50%	30.68	22.38	**27.1%**	164	164	5	672
AI-chat-powered-by-open-ai-api-react-node	1	0	1	0	0	71%	9.06	1.54	**83%**	82	82	5	688
API-Search	1	0	2	0	2	12%	13.04	13.14	-0.8%	196	207	6	852
Calculadora	0	0	1	0	2	66%	20.37	13.99	**31.3%**	52	50	5	800
CashTrackr	0	0	1	0	3	00%	14.15	14.94	-5.6%	153	164	5	852
CRUD-Context-React	0	0	4	0	2	22%	48.67	46.78	3.9%	235	235	6	792
hackaton-euraz-2023	0	3	4	0	0	12%	83.22	38.38	**53.9%**	765	740	8	831
hacker-news-app-2023	1	1	2	0	0	15%	12.98	9.45	**27.2%**	79	79	5	852
healthCare	13	14	39	0	0	23%	155.7	122.4	**21.4%**	919	964	5	672
honey-rae-repairs	10	4	22	0	3	93%	24.85	5.58	**77.5%**	416	284	6	835
hooksMixed-1	0	0	2	0	0	67%	5.81	4.21	**27.5%**	78	78	5	826
TypeWriter	0	1	37	0	4	22%	24.82	24.63	0.8%	616	730	13	703
farm-ng-core	0	0	3	0	0	68%	95.42	97.14	-1.8%	6	28	7	604
css-fx-layout	0	1	5	0	0	47%	24.65	24.93	-1.1%	50	65	6	1164
chicken-coop	2	3	7	0	0	54%	6.74	5.53	**18%**	172	164	7	605
Fokasu	0	0	1	0	1	57%	17.9	18.02	-0.7%	85	85	8	687
laser-shooting	0	0	3	0	0	00%	207.14	219.13	-5.8%	444	444	10	1098
csi-conference-frontend	0	2	13	0	0	01%	57.89	57.13	1.3%	230	351	7	623
baklava	0	0	1	0	0	00%	0.52	0.46	11.5%	3	3	11	700
TwitchKillMe	0	1	2	0	0	–	–	–	–	85	67	7	688
Average	**1.6**	**1.8**	**9.1**	**0.0**	**0.9**	**33.3%**	**40.25**	**34.08**	**20.54%**	**281.17**	**292.35**	**6.7**	**779.8**

This paper presents five anti-patterns that were identified by manually inspecting 40 repositories. This paper is the first work to characterize anti-patterns leading to needless re-rendering. In the absence of ground truth, it is difficult to gauge the exhaustiveness of the anti-patterns presented in this paper. However, we observed that these five anti-patterns are extremely prevalent, as 92.1% of the 7,760 repositories we studied exhibited at least one.

92.1% of the 7,758 studied React applications under consideration exhibit at least one anti-pattern.

6.3 RQ2: How often do the transformations cause behavioral differences?

We manually examined each application's behavior before and after refactoring to ensure the transformations did not introduce differences. We opted for a manual approach since UI-based applications are generally not well-tested, given the complexity of simulating human interaction in tests. The goal of our manual interaction is to simulate a simple interaction to exercise code affected by our transformations and not to emulate the full use of an application. To do so, we explored every page of the application, interacting with every interactive element, exercising all transformed code. A step-by-step account of these interactions appears in the appendix.

After careful analysis of all applications, we only observed one behavioral difference in the *TwitchKillMe* project. Upon inspection of the code, we observed the implementation was not aligned with React's development philosophy, having a nested component definition [27] and did not use props for parent-child communication. This is not self-contained and violates the locality of behavior.

We observed only one situation where *Reactor* caused behavioral differences in our 23 carefully studied subject applications.

6.4 RQ3: How do the transformations impact the number of re-rendering operations?

We instrumented each React component to log rendering operations as they were being performed, using the same scenarios that we used to answer RQ2, noting the total number of rendering operations before and after transformation.

Table 1 summarizes the results of this experiment. The **Red. %** column indicates the % reduction in total rendering operations during our manual analysis and the **Cyclomatic** columns show the cyclomatic complexity before and after transformation. We see significant reductions in the number of rendering operations in the majority of cases. In *honey-rae-repairs*, there is a large reduction as most of the application's functionality is related to forms that are transformed into uncontrolled components. *age-react* and *Fokasu* contain a timer causing some of the components and prop-less child components to re-render every second. Memoizing these prop-less child components ensures they are rendered only once.

Notably, for *CashTrackr*, we see no change in the number of rendering operations performed. There are two major reasons for this. First, the prop-less components are only rendered once. Second, one component in the application takes functions as props (which can be wrapped in `useCallback`) but also takes an array of "expenses". Due to JavaScript's highly dynamic nature, *Reactor* only performs intra-procedural analysis and cannot fully refactor this anti-pattern. In applications with at least one downloaded asset, we do not see any differences in the number of rendering operations for *baklava* and *laser-shooting*, as the propless component is either a top-level component or relies on an external routing library, which prevents local optimization.

In the 23 applications under consideration, the number of rendering operations is reduced by 33.3% on average after applying the transformations suggested by *Reactor*.

6.5 RQ4: What is the impact of the transformations on performance?

We used the Chrome Developer React Profiling Tools to collect the time spent rendering all components during each of the scenarios discussed in RQ3, reported in **Avg. Time Rendering** columns in Table 1. We also computed the % difference in time before and after refactoring, shown in column **% Time Saved** in the same table; bold entries in this column indicate statistically significant differences in render times before and after refactoring at 95% confidence using Welch's t-test (this test does not assume equal variance). Overall, we see improvements in render times in most cases, and no statistically significant negative impact to performance.

The performance impact of superfluous re-rendering becomes evident as an application scales. To illustrate, we collected an additional three applications that had a number of components that could vary dynamically: a spreadsheet, a drawing application, and a JSON editor. We configured these applications at small, medium, and large scales, and conducted experiments where we investigated the performance before and after refactoring at each scale using the Chrome Dev Tools React Profiler tab. These experiments were conducted on a 2023 MacBook Pro with an M2 Max chip and 64GB RAM. We manually confirmed that application behavior was preserved after refactoring, and these are all available in the artifact.

6.5.1 Spreadsheet. In this application [18], *every cell* of the spreadsheet was re-rendering any time a user either clicked on or entered data in *any cell*; one would expect rendering performance to become progressively worse as the sheet's size increases. *Reactor* refactors this application to avoid re-rendering unmodified cell. We prepared three configurations for this application: a small 20x20 spreadsheet, a medium 60x60 spreadsheet, and a large 100x100 spreadsheet.

In the small spreadsheet, typing one character into one cell incurred, on average, a 6.89ms re-render before refactoring, and a 2.07ms re-render after refactoring (3.3x improvement). In the medium spreadsheet, these re-renders took 43.4ms and 7.74ms on average, resp. (5.6x improvement). In the large spreadsheet, these re-renders took 125.35ms and 19.37ms on average, resp. (6.5x improvement). Before refactoring, we observed a noticeable lag in the medium scale experiment, and a significant lag in the large scale experiment. After refactoring, we noticed no lag in any case.

In terms of code complexity, the main code transformation was memoization of the component responsible for cells in the spreadsheet, and wrapping two functions passed to the cells with useCallback; together, these simple code changes result in up to a 100ms improvement in render time while typing in a cell in this spreadsheet, which we found to be noticeable.

6.5.2 Drawing App. In this application [11], users draw on a grid, and interacting with *any pixel* causes *all other pixels* to re-render; *Reactor* refactors this application to avoid re-rendering pixels that were not interacted with. We prepared three configurations for this application: a small drawing area of 32x32 pixels, a medium area of 64x64 pixels, and a large area of 128x128 pixels.

In the small configuration, drawing a single pixel caused a re-rendering operation lasting 14.51ms on average before refactoring, and 2.76ms on average after, an improvement of 5.3x. In the medium configuration, these re-renders took 51.83ms and 7.67ms on average, resp., an improvement of 6.8x. Finally, in the large configuration, these re-renders took 197.43ms and 26.49ms on average, resp., an improvement of 7.5x. We observed noticeable lag before refactoring in the large configuration, and no lag in any case after refactoring.

In terms of code complexity, the main code transformation was the memoization of the component responsible for pixels, and wrapping the function passed into the pixel components with useCallback. These are again very simple code transformations that result in significant improvements in application performance, in particular as the drawing grid becomes larger.

6.5.3 JSON Editor. In this application [5], users load a JSON and edit in in-browser, and save it to disk. A `Form` component from the `react-bootstrap` library is created for each element in the JSON. Before refactoring, the underlying JSON object was represented with a React state variable and a new JSON object was created each time *any form* was modified, causing *all forms* to re-render. *Reactor* refactored the application to manage the JSON with a ref instead, and the only component that changes when editing a form after refactoring is the form itself. We created small, medium, and large JSONs with 100, 1000, and 10000 elements resp. for this experiment.

Application performance is greatly improved by *Reactor*. Before refactoring, writing a single character into one field incurred a 17.05ms render on average with the small JSON, a 106.74ms render on average with the medium JSON, and a 1,035.88ms render on average with the large JSON; lag was noticeable with the medium JSON, and prohibitive with the large JSON. After refactoring, editing a form is seamless in all cases as no other components on the page re-render. In fact, HTML forms (which underlie the `react-bootstrap` form component) are managed a bit differently than other components as they maintain limited internal state[5], and modifying the contents of a form does not cause a change in the VDOM unless other components depend on the contents. In a sense, these forms render their own content without going through React. After refactoring, the only component on the page that changes when a form is edited is the form itself, and the React profiling tools register no other component renders; this results in React spending 0ms re-rendering components after refactoring.

The main change in this application was managing the JSON object with a ref, which required *Reactor* to change how this object was referenced in 8 code locations (refactoring references to

[5]See https://legacy.reactjs.org/docs/forms.html, and https://react-bootstrap.netlify.app/docs/forms/overview/ for more information.

`currentJson` into `currentJson.current`). These changes were all localized to a single file, and had a dramatic effect on performance.

The impact of needless re-renders manifests itself particularly as applications scale, and small, simple code changes reduces rendering time and can make applications more responsive.

6.6 RQ5: What is the impact of the transformations on code complexity?

We measured code complexity metrics such as sloc, cyclomatic complexity, and Halstead complexity. For brevity, we only present cyclomatic complexity in Table 1 but all metrics show a similar trend and are available in the Appendix. On average, we observe a slight increase in the Cyclomatic complexity of the programs from 281.17 to 292.35. Some projects such as *16-feb-react-grupo-1* and *css-fx-layout* show a small increase in cyclomatic complexity, whereas projects such as *honey-rae-repairs* and *chicken-coop* show a slight reduction in code complexity. Cyclomatic complexity can decrease, e.g., if the refactoring eliminates state setters (which eliminates function calls and callbacks being passed), or if components are made into uncontrolled components (which eliminates at least one change handler). Overall, the code transformations do not result in a significant increase in code complexity.

In the 23 applications, we observe only a slight increase in code complexity after applying the transformations suggested by *Reactor*.

6.7 RQ6: What is the cost of the analysis?

There are three components to *Reactor* 's cost: (i) building the CodeQL database, (ii) executing all queries, and (iii) applying the transformations. Table 1 reports (i) and (ii) under **DB Build Time** and **Total Query Time**, respectively. Transformation time is negligible (milliseconds), and DB build time is under 15 seconds in all cases. The most expensive step is query execution, averaging 13 minutes per application on a local machine with 16GB RAM, and never exceeding 20 minutes. We also ran the query step across all 7,758 repositories on a CentOS7 server with 128GB RAM to handle the larger workload, where it averaged 97 seconds per repository.

On average, it takes less than 13 minutes to transform one project.

7 THREATS TO VALIDITY

It is possible that our set of repositories do not represent all React applications, and the anti-patterns we identified may not cover all forms of re-rendering inefficiencies. To mitigate this, we randomly sampled 10,000 GitHub projects declaring React as a dependency, and selected subject applications from this pool. We further enriched our sample with repositories containing downloadable release assets to better capture real-world usage. Regarding the latter, our anti-patterns were derived from repeated manual investigations of components exhibiting superfluous re-rendering without visible UI changes. While this process may have missed rarer or more subtle inefficiencies, the patterns we identified were prevalent across the broader 7,758-project corpus, suggesting they capture important and recurring problems. Future work could uncover additional inefficiencies; Reactor's design supports extending it with new detection and transformation rules as needed.

To assess the impact of the transformations on the number of rendering operations, we needed to observe the dynamic execution behavior of the subject applications. Testing React applications is difficult, and many React applications lack test suites, which was also the case for all of our subject applications. Therefore, we manually interacted with the applications by entering text in form fields, clicking buttons, etc. These interaction scenarios may not represent how actual users interact with the applications, which may introduce bias in our results. To mitigate this threat, we carefully documented the interactions and included them in supplemental material to enable reproducibility. Furthermore, we ensured that the interactions exercised as much of the application as possible, visiting every reachable page of each application.

The code transformation technique presented in this paper draws inspiration from Turcotte et al. [29] and Gokhale et al. [16] and suffers from similar threats to validity. Concretely, the proposed code transformations are not guaranteed to preserve program behavior and are unsound. This unsoundness can primarily be attributed to unsoundness in the static analysis, which is inevitable due to the dynamic nature of JavaScript. Therefore, the proposed code transformations should be treated as suggestions and carefully reviewed by developers before application. In spite of this unsoundness, we found that *Reactor* proposed behavior-altering transformations in only one situation in our evaluation.

8 RELATED WORK

Related work can be grouped into four categories: Analyzing React and its semantics, optimizing UIs, general JavaScript performance optimizations, and work in progress on Meta's new React compiler.

Analyzing React and its Semantics. Anastasia et al. studied vulnerability reports from React.js, focusing on how third-party dependencies impact security, finding that that managing updates effectively can mitigate common vulnerabilities [6]. Ferreira et al. present an empirical study of refactoring in React applications, identifying a catalog of 25 refactoring operations tailored to React alongside 17 adapted traditional refactoring operations, providing actionable insights for enhancing the design and maintainability of React applications [9]. In [26], the authors explored the development principles and architecture of React, highlighting its component-based structure, one-way data flow, and functional programming principles that enhance performance and developer usability. Madsen et al. presented a formal semantics for a core subset of React, capturing the essence of its component life-cycle, state changes, and reconciliation [19].

UI Optimization. Optimizing UI frameworks is critical for interactive applications to ensure responsiveness and enhance user experience. Diniz-Junior et al. conducted a comparative study of three Java-Script UI frameworks: React, Vue, and Angular. They found that Vue had the fastest DOM manipulation, React had the best interaction time, and Angular had the largest bundle size [10]. Similarly, Siahaan et al. [25] conducted an analysis on the rendering performance of React, Vue, Next.js, and Nuxt.js. They found that Vue had the fastest initial rendering times, while React excelled in speed, index and user interactions. Next.js was best in server-side rendering. Additionally, Ollila et al. [22] compared rendering strategies of modern web frameworks, including Angular, React, Vue, Svelte, and Blazor. They noted frameworks like Vue and Svelte, which automatically track dirty components and only process data bindings on updates, perform significantly better than those like React and Blazor, which re-render entire subtrees and do not differentiate between static and dynamic content. These studies compare rendering performance of various JavaScript UI frameworks, whereas we focused on improving React's responsiveness.

JavaScript Performance Optimization. Addressing performance issues in JavaScript is crucial for optimizing web applications. Selakovic et al. [24] demonstrated that inefficient API usage falls among JavaScript's most important performance issues. They found that most optimizations require minimal code changes. Gokhale et al. [16] introduced Desynchronizer, a tool that optimizes JavaScript by automatically refactoring synchronous API calls to asynchronous ones using static analysis. Turcotte et al. [30] identified antipatterns in JavaScript's promises and async/await, leading to inefficiencies and developed DrAsync, a tool to detect and visualize these issues. Arteca et al. [7] demonstrated that sub-optimal scheduling of asynchronous I/O operations in JavaScript can be improved by reordering them using static side-effect analysis, resulting in significant performance gains. Ferreira et al. [9] provided a comprehensive catalog of React-specific refactoring by studying top GitHub projects, offering insights into best practices for maintaining and improving the quality and performance of React applications.

The React Compiler. Meta's React team recently announced the React Compiler [20], an experimental extension to React that aims to render React applications more efficiently. However, the compiler does not aim to provide perfectly optimal re-rendering with zero unnecessary computation. Their method introduces a new compilation layer, which inherently limits the range of optimizations due to the need to guarantee correctness at the compiler level. Furthermore, to leverage the complete set of features, the React Compiler requires users to use React version 19. Our approach only requires using functional components, the preferred paradigm since the introduction of life-cycle hooks in React 16.8.

We attempted to compare their approach with ours. However, React 19 introduces backward-incompatible changes that cause compilation failures in our subject applications. This suggests that, unlike our technique, it will require developers of existing React applications to manually upgrade their applications. Additionally, we encountered issues after migrating to React 19, where the React Profiling Dev Tools either broke or failed to display memoization badges, making performance comparison infeasible. This highlights the practical challenges of adopting their approach, while our method remains accessible across all React versions.

9 CONCLUSION

We have identified 5 anti-patterns that commonly give rise to unnecessary re-rendering in React applications. For each anti-pattern, a set of code transformations was proposed to reduce how often React components are re-rendered. To detect cases where needless re-rendering may occur, we have defined static analyses for detecting instances of the anti-patterns, which are used in a set of rewrite rules that produce suggestions on how affected code fragments can be transformed. The static analyses are potentially unsound, so these suggestions must be reviewed carefully by a developer to ensure that behavior is preserved. These analysis and rewrite rules were implemented in a tool named *Reactor*.

In a large-scale evaluation of 7,758 projects, we detected at least one instance of our defined anti-pattern in 92.1% of projects, indicating their prevalence in real-world applications. In an empirical evaluation of *Reactor* on 23 subject React applications, a total of 313 instances of the anti-patterns were transformed, and the number of rendering operations was reduced by 33.3% on average. We also observed that transformations result in a small increase in cyclomatic complexity from 281.17 to 292.35 on average. This relatively small increase in complexity leads to an average decrease of 20.54% in time spent rendering, and additional case studies show that application responsiveness improves significantly as the number of components scales up. Moreover, we observed only one instance of unsoundness in our experiments.

10 FUTURE WORK & GENERALIZABILITY

In two cases, *Reactor* could not automatically eliminate the anti-pattern. In CashTrackr, this is due to the intra-procedural nature of *Reactor*'s analysis. On the day20 page of 30days-30projects, two state variables are highly entangled, requiring more significant refactoring. In both cases, *Reactor* could be applied successfully after a modest amount of manual transformation. In future work, we plan to extend our analysis inter-procedurally and revise the rewrite rules to handle these cases automatically.

While our research contributions center on formalizing and remediating re-rendering anti-patterns using CodeQL and declarative rewrite rules, we note that simpler syntactic rules could potentially be adapted into ESLint plugins. Such extensions could help developers adopt our findings directly into existing everyday workflows.

Reactor is currently tailored to React, but the problem of superfluous re-rendering is not unique to it. Other popular front-end frameworks such as Vue and Angular also follow unidirectional data flow models and exhibit similar rendering behaviors. However, there are several fundamental differences between these frameworks. For example, React and Vue both use a Virtual DOM for change detection but have different implementations of it, whereas Angular uses Zone.js [4]. Both Angular and Vue use HTML templates and prefer to extend HTML with directives and pipes, whereas React extends JavaScript with HTML instead. As a result of these differences, the solutions to the same problem are different in each framework. As a concrete example, consider the case where a component should re-render only if some of its arguments change. To achieve this, React wraps the component in a call to `memo`, Angular uses the component configuration to set the `ChangeDetectionStrategy` to `OnPush`, and Vue would use the `v-memo` directive with the arguments as invalidation conditions. As such, it would be extremely difficult to develop a single solution that extends beyond a single framework, but the underlying ideas carry over. As future work, we plan to explore how our anti-pattern abstractions might be adapted to other popular frameworks, and whether automated transformations analogous to those in *Reactor* can be systematically derived.

11 DATA AVAILABILITY STATEMENT

We have made an artifact available at this Zenodo link.

REFERENCES

[1] . 2024. Build interactive web UIs using C#. . See https://dotnet.microsoft.c/..

[2] 2024. Calculadora. See https://github.com/Contimp/Calculadora/blob/master/src/App.js.

[3] 2025. escomplex. See https://github.com/escomplex/escomplex.

[4] 2025. Zone.js. See https://github.com/angular/angular/tree/main/packages/zone.js.

[5] AccessiTech. 2025. local-json-editor. See https://github.com/AccessiTech/local-json-editor..

[6] Terzi Anastasia and Bibi Stamatia. 2024. Managing Security Vulnerabilities Introduced by Dependencies in React.JS JavaScript Framework. In *2024 IEEE International Conference on Software Analysis, Evolution and Reengineering-Companion (SANER-C)*. IEEE, 126–133.

[7] Ellen Arteca, Frank Tip, and Max Schäfer. 2021. Enabling Additional Parallelism in Asynchronous JavaScript Applications. In *35th European Conference on Object-Oriented Programming (ECOOP 2021)*. Schloss-Dagstuhl-Leibniz Zentrum für Informatik.

[8] Citrusbug. 2024. React Statistics – An In-Depth Look at the Trends. See https://citrusbug.com/blog/react-statistics/#:~:text=In%202024%2C%20there%20are%20about,for%20their%20front-end%20framework..

[9] Fabio da Silva Ferreira, Hudson Silva Borges, and Marco Tulio Valente. [n. d.]. Refactoring React-Based Web Apps. *Marco Tulio, Refactoring React-Based Web Apps* ([n. d.]).

[10] Raimundo NV Diniz-Junior, Caio César L Figueiredo, Gilson De S Russo, Marcos Roberto G Bahiense-Junior, Mateus VL Arbex, Lanier M Dos Santos, Raimundo F Da Rocha, Renan R Bezerra, and Felipe T Giuntini. 2022. Evaluating the performance of web rendering technologies based on JavaScript: Angular, React, and Vue. In *2022 XVLIII Latin American Computer Conference (CLEI)*. IEEE, 1–9.

[11] Emily Doherty. 2025. Pixelsketch. See https://github.com/emilydoh/Pixelsketch.

[12] Evan You, Vue.js developers. 2024. cybernetically enhanced web apps. See https://vuejs.org/..

[13] Fabio Ferreira and Marco Tulio Valente. 2023. Detecting code smells in React-based Web apps. *Information and Software Technology* 155 (2023), 107111.

[14] GitHub. 2024. See https://github.com/github/codeql..

[15] GitHub. 2024. Prevalence of React. See https://github.com/facebook/react/network/dependents. Accessed Jun 07 2024..

[16] Satyajit Gokhale, Alexi Turcotte, and Frank Tip. 2021. Automatic migration from synchronous to asynchronous JavaScript APIs. *Proceedings of the ACM on Programming Languages* 5, OOPSLA (2021), 1–27.

[17] Google. 2024. See https://angular.io/..

[18] Agnieszka Goralczyk. 2025. react_spreadsheet. See https://github.com/ajgoralczyk/react_spreadsheet.

[19] Magnus Madsen, Ondrej Lhotak, and Frank Tip. 2020. A Semantics for the Essence of React. In *European Conference on Object-Oriented Programming*.

[20] Meta Platforms, Inc. 2024. React Compiler. See https://github.com/facebook/react/tree/main/compiler..

[21] Meta Platforms, Inc. 2024. React: The library for web and native user interfaces. See https://react.dev/..

[22] Risto Ollila, Niko Mäkitalo, and Tommi Mikkonen. 2022. Modern web frameworks: A comparison of rendering performance. *Journal of Web Engineering* 21, 3 (2022), 789–813.

[23] Rich Harris. 2024. See https://svelte.dev/.

[24] Marija Selakovic and Michael Pradel. 2016. Performance issues and optimizations in JavaScript: an empirical study. In *Proceedings of the 38th International Conference on Software Engineering*. 61–72.

[25] Mangapul Siahaan and Ryan Kenidy. 2023. Rendering performance comparison of react, vue, next, and nuxt. *Jurnal Mantik* 7, 3 (2023), 1851–1860.

[26] CACM Staff. 2016. React: Facebook's functional turn on writing Javascript. *Commun. ACM* 59, 12 (dec 2016), 56–62. https://doi.org/10.1145/2980991

[27] React Dev Team. 2024. Nesting and organizing components. See https://react.dev/learn/your-first-component#nesting-and-organizing-components. Accessed Oct 15 2024..

[28] Alexi Turcotte, Mark W Aldrich, and Frank Tip. 2022. Reformulator: Automated refactoring of the n+ 1 problem in database-backed applications. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. 1–12.

[29] Alexi Turcotte, Satyajit Gokhale, and Frank Tip. 2023. Increasing the Responsiveness of Web Applications by Introducing Lazy Loading. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 459–470.

[30] Alexi Turcotte, Michael D Shah, Mark W Aldrich, and Frank Tip. 2022. DrAsync: identifying and visualizing anti-patterns in asynchronous JavaScript. In *Proceedings of the 44th International Conference on Software Engineering*. 774–785.

[31] W3Techs. 2024. Comparison of the usage statistics of React vs. Vue.js vs. Angular for websites. See https://w3techs.com/technologies/comparison/js-angularjs,js-react,js-vuejs..