Proactive Debugging of Memory Leakage Bugs in Single Page Web Applications

Arooba Shahoor, Satbek Abdyldayev, Hyeongi Hong, Jooyong Yi, *Member, IEEE*, and Dongsun Kim, *Member, IEEE*

***Abstract*—Developing modern web applications often relies on web-based application frameworks such as React, Vue.js, and Angular. Although the frameworks accelerate the development of web applications with several useful and predefined components, they are inevitably vulnerable to unmanaged memory consumption as the frameworks often produce monolithic web pages, so-called, Single Page Applications (SPAs), in which no page refresh actions are made during navigation. Web applications can be alive for hours and days with behavior loops, in such cases, even a single memory leak in an SPA can cause performance degradation on the client side. However, recent debugging techniques for web applications focus on memory leak detection, which requires manual tasks and produces imprecise results, rather than proactively repairing memory leaks. We propose LEAKPAIR, a technique to proactively repair memory leaks in SPAs rather than following a classical and reactive debugging process. Given the insight that memory leaks are mostly non-functional bugs and fixing them might not change the behavior of an application, the technique is designed to proactively generate patches to fix memory leaks, without leak detection, which is often heavy and tedious. Thus, the proactive technique can significantly reduce the time and effort necessary to fix the memory leaks. To generate effective patches, LEAKPAIR follows the idea of pattern-based program repair since the automated repair strategy shows successful results in many recent studies. We extensively evaluate the technique on 60 open-source projects without using explicit leak detection. The patches generated by our technique are also submitted to the projects as pull requests (PRs). The results of PRs show that LEAKPAIR can generate effective patches to reduce memory consumption that are acceptable to developers. In addition, we execute the test suites given by the projects after applying the patches, and it turns out that the patches do not cause any functionality breakage; this might imply that LEAKPAIR can generate non-intrusive patches for memory leaks. Furthermore, we compare the performance of LEAKPAIR with that of GPT-4 as recent studies show that large language models are successful with program repair tasks. Our results show that our technique outperforms the language model.**

***Index Terms*—Memory leaks, program repair, non-intrusive fixes, single page web applications, proactive debugging.**

I. INTRODUCTION

MPAs (Multiple Page Web Applications) were the most popular architectural style until 2010 when building web applications. In MPAs, each page had to re-fetch and reload the entire web page for each user request. The traditional MPA approach incurs a longer page switch time owing to the server round-trip for each request, and this delay increases with the size and complexity of the server APIs. The burgeoning usage of smartphones and mobile apps and the growing demands for swift and responsive web apps inspired the web development community to change how web pages were architected and rendered.

To address the responsiveness of web pages, the concept of Single Page Applications (SPAs) was introduced as a new architectural style for web applications; this idea was first implemented by AngularJS, whereby rather than updating the entire webpage, only the data of the same page was updated [2]. In SPAs, instead of re-fetching and loading entire pages from the server upon each request, just the data (usually in JSON format) can be retrieved asynchronously from the server and inserted dynamically into the application, thereby preventing page reloads on navigation and data fetch requests [3]. Today, almost all contemporary social media apps make use of this architecture [4].

SPAs, however, are vulnerable to memory bloating due to their architecture in contrast to MPAs. Literally, SPAs maintain a single web page for a specific application, and all objects reside on that single page. Therefore, SPAs inevitably rely on the garbage collectors of browsers to manage the memory space.

Received 25 November 2024; revised 23 April 2025; accepted 4 May 2025. Date of publication 16 May 2025; date of current version 18 July 2025. This work was supported in part by the National Research Foundation of Korea (NRF) through Korea Government (MSIT) under Grant RS-2021-NR060080 and Grant 2021R1I1A3048013, in part by the Institute for Information & Communications Technology Planning & Evaluation (IITP) through Korea Government (MSIT) under Grant RS-2024-00437306 and Grant RS-2023-00222830, and in part by ICT Creative Consilience Program through the Institute of Information & Communications Technology Planning & Evaluation (IITP), Korea Government (MSIT) under Grant IITP-2025-RS-2020-II201819. Recommended for acceptance by Y. Xiong. *(Corresponding author: Dongsun Kim.)*

Arooba Shahoor is with WithPlaybook, Inc., Seoul 06307, Republic of Korea (e-mail: arooba.shahoor@gmail.com).

Satbek Abdyldayev, Hyeongi Hong, and Jooyong Yi are with the Department of Computer Science and Engineering, Ulsan National Institute of Science and Technology (UNIST), Ulsan 44919, Republic of Korea (e-mail: satbek@unist.ac.kr; ghdgusrl2000@unist.ac.kr; jooyong@unist.ac.kr).

Dongsun Kim is with the Department of Computer Science and Engineering, Korea University, Seoul 02841, Republic of Korea (e-mail: darkrsw@korea.ac.kr).

Digital Object Identifier 10.1109/TSE.2025.3571192

```
15  export default class SidePane extends React.Component<SidePaneProps, Sid
16      private div = React.createRef<HTMLDivElement>();
17
18      constructor(props: SidePaneProps) {
19          super(props);
20          this.state = {
21              currentPane: this.props.plugins[0],
22          };
23
24          window.addEventListener('hashchange', this.updateStateFromHash);
25      }
26
```

(a) Event listener memory leak in Rooster JS.

```
23
24          window.addEventListener('hashchange', this.updateStateFromHash);
25      }
26
27      componentDidMount() {
28          this.updateStateFromHash();
29      }
30
31  +   componentWillUnmount() {
32  +       window.removeEventListener('hashchange', this.updateStateFromHash);
33  +   }
```

(b) Patch for the memory leak in (a).

Fig. 1. Memory leak in Rooster JS [1] and its corresponding patch.

Each browser's JavaScript engine implements its garbage collector that is responsible for identifying and reclaiming memory occupied by objects that are no longer reachable from the program. However, there is still a high likelihood of unnecessary objects lingering around that do not get garbage-collected due to some unintentional reference, leading to memory leaks. Such leaks might not be a problem in MPAs, where on each page navigation, the page refreshes, clearing all the heap. In SPA, however, such leaks can easily accumulate to several megabytes as a single page remains alive for several hours or even days.

Because such memory-leaking patterns are not syntactically or semantically invalid code, browsers run the program without throwing any errors, and they go unnoticed in functional testing as well [5]. Consider the syntactically and semantically correct code scenario in Fig. 1(a) from Microsoft's `roosterjs` library [6]. Based on the React framework, the class adds a listener for a `hashchange` event (an event that is fired every time the part of the URL after the hash changes [7]), to each new instance of the class, without ever removing the listener, even after the component unmounts from DOM. This created a memory leak in the application.

An important point to note in the above scenario is that if the listener handler was attached to a local element that does not have references to any other object, it would have been automatically cleaned up by the garbage collector (GC) once the class instance was destroyed. In the above case, however, the event is attached to the root node (window object), which the GC never cleans up, even after the instance is destroyed. A simple fix to this memory leak was applied by the project developers (Fig. 1(b)) by explicitly removing the event in the component destructor function.

There have been a limited number of studies [8], [9], [10], [11], [12] on the problem of memory leak detection in the web domain. These studies focus on automating the detection of memory leaks, the most relevant and notable of which is BLeak [12], which is an automated memory leak detection tool for client-side web applications. BLeak requires a scenario file written by the users to run the app in a loop in a headless browser and takes around 10 minutes to execute. The details of other studies will be presented later in the Related Work section.

We present LeakPair, an approach to generating patches that repair memory leaks in SPAs. Unlike typical automated program repair approaches, LeakPair can be applied without requiring bug locations or relying on leak-detection techniques. It automatically detects code snippets that can potentially cause memory leaks and fixes them using non-intrusive (i.e., functionality-preserving) transformation rules we mined from existing code.

While test-driven program repair [13], [14], [15], [16] (also known as generate-and-validate repair [17]) begins to work once a bug is detected by test cases, proactive program repair first applies patches to potential buggy locations. Then, a proactive approach measures a difference in properties (such as memory consumption and execution time) before and after applying the patches. The difference is provided as evidence of repair instead of validating patches by test cases, which is done in test-driven program repair after generating patches. Thus, proactive repair is a special kind of program repair approach.

In summary, this paper contributes the following:

- **Initial Study Contributions:** In our earlier work presented at ASE 2023 [18], we introduced LeakPair, the first approach that fixes memory leaks in SPAs. In that work, we showed that for this targeted problem, a simple pattern-based approach can be effective, as evidenced by the developers' acceptance of the patches generated by LeakPair. Conceptually, we introduced the idea of proactive repair, which unlike typical automated program repair approaches, performs repairs proactively before a problem occurs.
- **Initial Study Technique:** The study follows a three-step process; we first mine the common memory leak patterns in applications developed using Angular [19] and React [20] (two of the most widely used SPA frameworks), and the corresponding fix patterns from `GitHub` and `StackOverflow`. We then develop a CLI tool that parses the given project, traverses the AST (Abstract Syntax Tree) to detect the leak patterns, and fixes them with the corresponding fix patterns. Finally, the tool is evaluated on subjects with fixed memory leaks (to compare LeakPair's fixes with those of the developers), as well as on new subjects. The tool successfully replicates the fixes for the 19 known leaks and repairs 18 previously unknown leaks, the patches for which are submitted as PRs to the developers.
- **Extended Study Contribution:** Our focus in this extended study is on improving the generalizability of our approach. To achieve this purpose, we extend the application of LeakPair to Vue (another widely used SPA framework). As a result, our technique is extensively evaluated on 60 open-source SPAs, each based on either React, Angular, or Vue. In addition, we increase the iterations for

the memory footprint measurement from 10 (original) to 25 rounds for each subject to ensure the soundness of the evaluation results.

In addition to the extensive evaluation, we compare LEAKPAIR with the results generated by GPT-4 [21] as recent LLMs perform very well for program repair tasks [22]. This study not only explores the effectiveness of GPT-4 in fixing memory leaks of SPAs, but also compares the qualitative aspects of patches generated by LEAKPAIR and GPT-4. The results show the limitation of LLMs when addressing the memory leak repair task. To the best of our knowledge, this is the first study to compare the effectiveness of GPT in fixing memory leaks.

We also present a brief case study of the SoundCloud memory leak, underscoring the severity and prevalence of memory leaks. Furthermore, to clarify why the current state-of-the-art in memory leak diagnosis falls short, we have incorporated a comprehensive section that delves into the heap snapshot analysis technique. This segment provides a detailed examination of the "3-snapshot technique", and demonstrates its limitations, substantiating the necessity for advancements in this area.

The paper also revisits the results of the live study on pull requests, aiming to identify potential factors influencing the approval or non-approval of automatically generated patch submissions. In addition, we investigate the case of intrusive memory leak repair, particularly the case of automated repair of memory leaks caused by global collections (arrays, sets, etc.), which can break functionality. We explore how and why such collections cause memory leaks, the rationale behind our tool not being able to repair them non-intrusively, and present potential solutions to address these edge cases.

Lastly, this paper incorporates an expanded discussion of related works, offering a more holistic view of the present landscape of automated program repair, memory leak debugging, and proactive patch generation.

- **Extended Study Technique:** Following the approach in the prior study, we curate three memory leak (and fix) patterns for Vue (FP5a, FP5b, FP5b in Section III-C). For this purpose, 11 Vue subjects with known leaks and 12 subjects where LEAKPAIR repaired unknown leaks are added. The patches for the previously unknown leaks are submitted as pull requests, as was done in the original study. The tools and techniques for measuring memory footprints remain unchanged.

  For exploring the case of intrusive patches and gauging the impact of the intrusion, we follow the evaluation process of our first study; we automate the patch generation based on the curated fix pattern and run the SPA containing the global-collection leak along with the provided test suite, both before and after the patch application.

The remainder of this paper is organized as follows. After illustrating the background and motivation of this study in Section II, we propose our approach, LEAKPAIR, in Section III. Section IV empirically evaluates our approach and reports on the experiment results. Section VI discusses the comparison of LEAKPAIR against the state-of-the-art program repair technique, and intrusive patches for memory leaks. After surveying the related work in Section VII, we conclude with directions for future research in Section VIII.

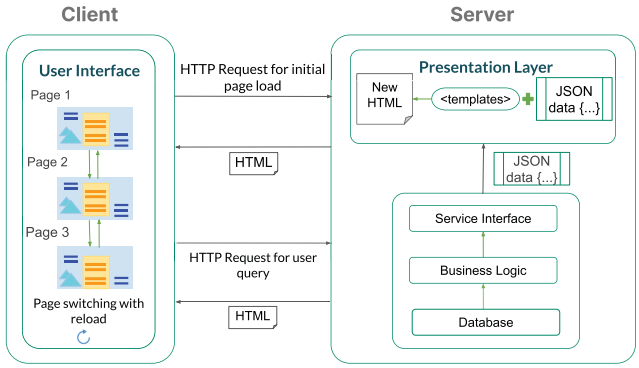

Fig. 2. Overview of multi-page application (MPA) architecture [3], [4].

II. BACKGROUND AND MOTIVATION

A. Single Page Web Applications (SPAs)

This section compares Multiple Page Applications (MPAs) and Single Page Applications (SPAs) and discusses why SPAs are vulnerable to memory leaks.

In MPAs, the actions taken by the user on the webpage (such as navigation, form submissions etc.) trigger HTTP requests to the server as shown in Fig. 2. The server retrieves data from data sources, merges them with server-side templates, and then sends the fully rendered HTML (page) to the client for display. This results in a page refresh for each such user interaction. In addition, the user session and data are persisted on the server; any time the session state or data need to fetched or updated, the server has to be queried, and the client (and the user) waits for the update to be completed on the server, resulting in a poor app responsiveness [3].

In SPAs (Fig. 3), while there may be multiple JavaScript, CSS, and other resource files, there is typically a single HTML file that serves as the initial entry point for the application. Within this single HTML file, templates are defined by SPA frameworks (Angular, Vue, etc.). These templates provide placeholders where data can be dynamically inserted or interpolated. Now, instead of the server generating fully-rendered HTML pages, it only serves data (often in JSON format) through APIs or other endpoints. The client then retrieves this data and dynamically merges it with the templates to generate the final 'views'. Each such dynamically generated view represents a distinct 'page' the user interacts with; the difference lies in the smooth transitioning of these pages, as there are no page reloads during the process. The logic of merging the data with the right template, routing to the right view, and maintaining the life cycle of a single view are all defined using the SPA frameworks [4].

In addition, an SPA caches all the received data from the server so that the user is still able to interact with the app in

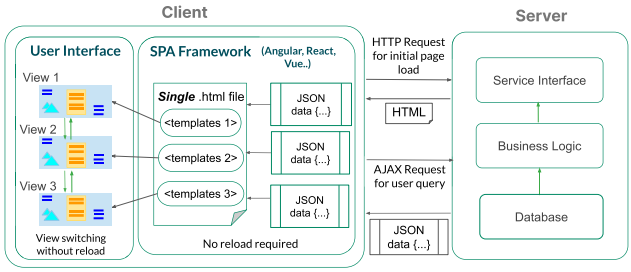

Fig. 3. Overview of the Single-Page Application (SPA) architecture.

Fig. 4. Memory leak detection in fiit website using heap snapshots.

case of poor connection or connection loss, and any new data can be synced once the connection improves/restores [3].

B. Garbage Collection and Memory Leaks in SPAs

In MPAs, memory leakage may not be a critical issue since the web pages are switched frequently and, as the browser switches to a new page, the memory reserved by the previous page is reclaimed by the garbage collector. Most modern web apps, however, are single-page apps that update the content without switching the web page. This means that a single web page can be active for several hours or even days [10]. When memory leakage in such applications accumulates over time, it not only slows the program execution and causes data processing latency but may also lead to program crashes and incompatibility with other applications.

Several existing popular websites (including the libraries they use) suffer from memory leakage that adversely affects the responsiveness of the browser. In its blog post [23], SoundCloud discusses its web application built with React and Redux that suffered a memory leak (Redux is a state management library that stores and organises data in a central location called the 'store' [24]). The leak stemmed from the Redux store continuously growing with each request, as objects accumulated in the store without removal, even when no longer needed. While React typically handles garbage collection when components unmount, Redux maintains references to all data, preventing its release. The leak was fixed by implementing a custom Garbage Collector on top of the Redux store, that ensures that only the necessary entities required by mounted components remain in the store.

Vilk and Berger [12] reported that more than 99 percent of Google Chrome crashes on low-end Android phones are the result of memory issues. They also identified more than 50 memory leaks in popular applications, including JavaScript frameworks, and Google applications. Another leak detection study [25] revealed public-facing SPAs leaking up to 186 MB per interaction.

As will be demonstrated in the next section, since such leaks are hard to discover and diagnose, developers rather choose to invest their time and effort in addressing more 'apparent' application issues. Finally, oftentimes developers may wrongly attribute the lagging app behavior to the user's browser, internet connection, or even their systems.

C. Automated Efforts and State of the Art

Unlike manually managed languages (such as C and C++), the JavaScript standard (ECMAScript) does not provide any interface for developers to monitor the memory usage of the app or manipulate the Garbage Collector, which makes diagnosing the leaking memory a cumbersome task for the developers [26]. Consider testimonials [27], [28], [29] as well as the following comments from SPA developers on Github and StackOverflow regarding the obscure and evasive nature of memory leaks and their detection:

> *I looked at the Chrome Dev Tools and taking heap snapshots to see if there is an increase in memory and it is apparent that there is when I see the memory shoot from 123MB to 200+MB after a few actions within the application. Now this is a good tool for determining whether there is a possible memory leak or not, but it's absolutely hard to read and understand, which doesn't help me determine where the issues lie [30].*
> *This issue has been around for nearly 3 years now. (I usually don't like to start a message this way unless I tried something to fix the issue myself... Which I did here! and failed miserably as it seem quite complex to get to the bottom of it...[31].*

In order to address memory leak issues, the root cause needs to be diagnosed first. Although there have been automated techniques and approaches to detect memory leaks in web applications [8], [9], [10], [11], [12], these techniques have several limitations, including (1) dependency on the browser's heap snapshots, (2) non-trivial effort required for writing a test-driver script and (3) imprecision. The state-of-the-art for memory leak detection in SPAs (and websites in general), hence, is the manual analysis of heap snapshots via the browser dev tools.

The three-snapshot technique was first introduced by Loreena Lee and the Gmail team in 2012, to address leaking memory issues in Gmail [32]. The workflow is as follows:

First, capture the heap snapshot at the start of the application load, then, interact with the application, take the second snapshot, followed by the same user actions as taken before, and finally take the last snapshot. Then, in the Summary view of heap snapshot 3, perform a comparison between snapshots 1 and 2 to filter the objects allocated between the 2 snapshots. Finally, use the Retainer view to see what is referencing these objects in order to find the leaking object source.

Fig. 4 is a screenshot from one of the web posts of fiit (UK's #1 rated fitness app) [33], where the developer shared their experience of debugging memory leaks on the official website, using this technique.

This kind of diagnosis, however, is not always accurate as the actual leak may appear low on the heap snapshot list and may even have a small retained size, making it an unlikely target for investigation. Furthermore, reading through the heap snapshot content can be highly time-consuming without gaining any valuable information. The JS engine for browsers organizes the memory consumed by the web app in a graph of nodes and edges [34]. The heap snapshot is a flattened version of this graph, in a JSON format, which, in addition to the actual objects of the web application, includes meta-data about the format of the memory graph and the shape and size of every object contained in the memory graph data (including internal JS engine objects). Moreover, unnamed objects are frequent, as JavaScript is dynamically typed, and web page source code is minified and obfuscated to reduce the size of the JavaScript code. This leaves too much noise for the user to be able to drill down to the actual unreachable objects that are leaking memory.

D. Non-Intrusive Repair Without Replicating Actual Memory Leaks

Non-intrusive patches: Our intuition here is to apply non-intrusive patches [35] to all potential memory leaks. If the patches are non-intrusive (i.e., behavior-preserving), it is not necessary to detect memory leaks before repairing them. As the patches do not change the behavior of a target program, it is better to repair as many (potential) leaks as possible, which eventually improves the maintenance quality. Such patches are unlikely to introduce new functional bugs and are often easy to understand. The tradeoff for developers is obvious: applying these patches is beneficial as they are simple and non-intrusive. Avoiding the leak detection step is a huge advantage, as this step is tedious and time-consuming due to the dynamic analysis involved. A similar approach was used in [35] to fix performance bugs. However, to the best of our knowledge, ours is the first work to proactively repair memory leaks directly in the source code of web applications.

Pattern-based program repair: To fix the memory leak issues, we employ pattern-based program repair. While we considered other types of program repair techniques as well, they were found to be less suitable for fixing memory leaks proactively. Most existing APR techniques (e.g. [13], [15], [36]) are test-driven, meaning that they require a test suite to drive the search for a patch, while we do not assume the existence of such a test suite. Note that recent neural program repair techniques (e.g. [37], [38]) also require a test suite to validate the generated patches. The issue of the trustworthiness of the generated patches is also a concern for such techniques.

In comparison, we curate fix patterns that are likely to be non-intrusive and apply them to the potential memory-leak locations of the program. Our pattern-based program repair can also be viewed as a static-analysis-based repair similar to FOOTPATCH [39] and SAVER [40], tools fixing the memory leaks of C/Java[1] programs — we statically detect potential memory-leak locations and fix them. These techniques typically involve substantial efforts by both tool developers and users to enable static analysis. For example, SAVER requires the semantic models for libraries to perform static analysis and fixing. By contrast, our pattern-based approach does not involve any heavyweight analysis and can be readily applied to any SPA program. As will be shown in Section V-B, the patches generated from LEAKPAIR are often accepted by real-world developers, demonstrating the practical value of our approach.

[1]SAVER cannot handle Java programs.

III. LEAKPAIR

A. Overview

Our approach, LEAKPAIR, consists of two steps: (1) fix pattern mining, and (2) memory leak repair using the fix patterns. In the first step, we manually examine program patches or pull requests addressing memory leaks, together with commit messages, code reviews available in open-source projects, and Q&A posts. After identifying common and recurring fix patterns from the patches, we implement an edit script for each pattern, which can generate non-intrusive patches. In the second step, we scan a target project (i.e., SPAs) to apply our fix patterns. Each fix pattern can naturally specify which data or object types are associated with it. A corresponding edit script can then be applied accordingly. Each pattern changes all locations, where applicable, in the target project.

B. Mining Fix Patterns for SPA Memory Leaks

Since our goal is to identify recurring common patterns of memory leaks and their corresponding patches in SPAs, we first collect the most common leaks available publicly, by using specific keyword search on `GitHub` and `Stackoverflow`. Then, we carefully extract common patterns of leaks and their corresponding patches. Obviously, this is a manual task and is time-consuming. Nonetheless, numerous previous studies [14], [15], [16], [35], [41], [42], [43] have demonstrated that this strategy is effective and useful, as we can reuse the fix patterns many times once they have been identified.

Two authors were part of this manual analysis. We use the following search process to collect issues and discussions relevant to memory leaks: (1) For Stack Overflow, we search through 1,000 posts whose titles, comments, or discussions contain a combination of two or more of the following keywords: ‘leak’, ‘memory usage’, ‘memory leak’, ‘memory’ and ‘React’, ‘Vue’, ‘Angular’ (depending on the framework). (2) For GitHub, we search through 1,000 commits, PRs, issues, and discussions containing any of the above keywords as labels.

After investigating the search results, we collect leak patterns as per the following procedures: (1) We select common memory leaks reported at least five times across `GitHub.com` and `stackoverflow.com`, (2) the leaks should be acknowledged as valid, by at least two developers, (3) we further narrow down the leaks, which can be reproduced and tested locally, and (5) five leak patterns were selected, which are applicable to SPAs.

For each leak pattern identified in the previous step, we select fix patterns by looking at their original answers (for StackOverflow) or discussions (for GitHub). For each leak type, we extract, as fix patterns, the common fix suggestions in Stack Overflow that are accepted as the answer in at least two separate posts. From the leak patterns found in GitHub commits, we select the patches that were approved and merged in at least two separate projects. Among the above-selected fix patterns, we further filter the patterns based on their applicability to SPA projects.

All identified fix patterns are supported by examining actual memory footprint changes. We compare the memory footprints of revisions `before` and `after` applying the patches. If there were no differences between before and after memory footprints, we discard the fix patterns. We examine the memory footprints of patches applied to SPAs using MemLab [44].

C. Fix Patterns

As already discussed in Section II-B, the general root cause of memory leaks in SPA is an unused object that lingers in memory due to some unwanted reference that was not explicitly cleared by the developers. Hence, the fix for such leaks generally involves cleaning up any unwanted references to objects that have the potential to be retained in memory. In the SPA domain, this needs to be done when a `component` unmounts from the DOM (in the component destructor).

Following the procedure in Section III-B, we identified 7 fix patterns for generating non-intrusive patches for repairing memory leaks in SPAs:

FP1. Unreleased Subscription. In reactive JavaScript (RxJS), an `Observable` is a lazily evaluated computation that can synchronously or asynchronously return zero to (potentially) infinite values from the time it is invoked (subscribed) [45]. This indicates that they can keep outputting values even after the component is destroyed/unmounted, unless we explicitly tell them to stop. This means each time the component containing that subscription is rendered, a new observable is created in addition to the old one, because we never explicitly unsubscribed from the previous one. The stale data keeps getting piled up, never getting garbage collected, creating a memory leak.

In practice, developers may not always be able to figure out whether the observable they are subscribing to, is finite or infinite, and in these cases, it is best to explicitly `unsubscribe` when the component unmounts/destroys, just to be safe. This ensures that the Subscription is closed (if it was not already) and that proper cleanup is carried out. Nothing else will happen if it was previously closed.

Fix: The `takeUntil()` operator allows a notified `Observable` to emit values until a value is emitted from another Observable [46], i.e., the `takeUntil()` operator completes the stream it is attached to, when an Observable provided to itself, emits a value. Thus, if we provide another observer `O2` (see pseudo-code below) as input to the `takeUntil()` operator, and in the destructor we make `O2` emit a value (using the `next()` and `complete()` methods), that will clear the subscription and thus prevent the memory leak[2].

FP1. Unreleased Subscription

```
// O: Observer
// D: Destructor method

- O1.subscribe(() => {...})
+ O1.pipe(takeUntil(O2)).subscribe(() => {...})
...
+ D() {
        ...
+       O2.next()
+       O2.complete()
+ }

// Example
{
+ private ngUnsub = new Subject();
  ...
  this.userService.getLocal()
+  .pipe(takeUntil(this.ngUnsub))
   .subscribe(usr => {this.user = usr;});
  ...
+ ngOnDestroy(): void {
+   this.ngUnsub.next();
+   this.ngUnsub.complete();
+ }
}
```

In the above example, the `getLocal` method is called on the class field userService, which returns an observable. The observable is then subscribed to, and upon receiving results, the returned user object (usr) is assigned to the user instance variable. The fix involves the insertion of `pipe(takeUntil(this.ngUnsub))` which ensures that subscriptions are cleaned up properly. By using `takeUntil(this.ngUnsub)`, the subscription to `this.userService.getLocal()` is automatically unsubscribed (i.e., completed) when `this.ngUnsub` emits.

FP2. Unremoved Event Listener. The notion of *retaining paths* is critical for finding the root cause of a memory leak. A retaining path is a chain of objects that prevents the garbage collection of the leaking object. The chain starts at a root object, such as the global object of the **main window**. The chain ends at the leaking object.

Active event listeners will prevent all variables captured in their scope from being garbage-collected. Once added, the event listener will remain in effect until (1) it is explicitly removed with `removeEventListener()` or (2) the associated DOM element is removed.

Fix: Unregistering the event listener once the SPA component unmounts/destroys, by creating a reference pointing to the event handler `H` (see pseudo-code below) and passing it to `removeEventListener()` method[3].

[2]https://github.com/blackbaud/skyux/pull/376/files
[3]https://github.com/microsoft/roosterjs/pull/921/files

FP2. Unremoved Event Listener

```
// T: Event target
// Y: Event type
// H: Event handler method
// D: Destructor method

T.addEventListener(Y, H)
...
+ D() {
        ...
+       T.removeEventListener(Y, H)
+ }

// Example
{
  ...
  document.body.addEventListener('touchend',
       this.handleMouseUp);
  ...

+ componentWillUnmount() {
+ document.body.removeEventListener('touchend',
     this.handleMouseMove);
+ }
}
```

FP3a. Uncleared Timeout Event

```
// I: Unique Timer ID
// D: Destructor method

- setTimeOut(() => {...})
+ I = setTimeOut(() => {...})
...
+ D() {
        ...
+       clearTimeOut(I)
+ }

// Example
{
+ caretPositionTimeout: number;
  ...
- setTimeout(() => { setCaretPosition(el, caretPos); },
     1000);
+ this.caretPositionTimeout = setTimeout(() => {
     setCaretPosition(el, caretPos); }, 1000);
  ...
+ componentWillUnmount() {
+  clearTimeout(this.caretPositionTimeout);
+ }
}
```

The example registers a listener on the touchend event provided by the mobile browser API. The touchend event is triggered when a finger or touch point is lifted off the touch surface on devices like smartphones and tablets. The example code assigns a handler function (handleMouseUp) that will be called whenever this event occurs.

A memory leak can occur if the developer does not unregister the event listener when the user leaves the page view. The fix is to remove the event listener in the componentWillUnmount function, which acts like a destructor in the React framework. This function is called just before the component (view) is unmounted from the DOM.

FP3a. Uncleared Timeout Event. The `setTimeout()` method executes a function or specified piece of code once the specified timeout value is reached. When any object is tied to a timer callback, it will not be released until the timeout happens. In certain scenarios, the program's logic requires the timer to reset itself; this causes it to run forever, thereby retaining the references of all the enclosing objects and disallowing the garbage collector to remove the memory. Even if the developers explicitly clear the `setTimeout()` in code conditionally, there is no guarantee it also caters for situations where the user navigates away after the `setTimeout()` is triggered but before the specified timeout value is reached.

Fix: Because each `setTimeout()` has its own memory reference, we must clear each one individually, using the `clearTimeout()` method, passing it the ID returned from the `setTimeout()` call (which uniquely identifies each `setTimeout()` reference). The patch involves clearing the timeout method just before the component is about to unmount from DOM i.e in the component destructor[4].

In the given example `setTimeout` is used to execute the custom `setCaretPosition` function after 1000 milliseconds (1 second). However, if the component is unmounted before the timeout completes, the callback may still try to execute, potentially leading to a memory leak.

The fix involves storing the timeout ID returned by setTimeout in a class property (`this.caretPositionTimeout`). This allows us to reference this specific timeout later. Then, just before the component unmounts (in the `componentWillUnmount` method), the `clearTimeout` function is called with the stored timeout ID (`this.caretPositionTimeout`). This cancels the timeout if it is still pending, preventing the callback from executing after the component has unmounted.

FP3b. Uncleared Interval Event. The `setInterval()` method repeatedly calls a function or executes a code snippet, with a fixed time delay between each call. Even after the component is unmounted from the DOM, the setInterval timer will keep on ticking (unless we explicitly clear the interval in the code), trying to update the state of a component that's effectively gone, thereby causing memory leakage [47]. Even if the developers clear these interval functions in the code on some condition, there is no guarantee that the clearing method will get a chance to execute before the user navigates away.

Fix: Each interval has a separate reference in memory, so we need to clear each individually, using the returned ID from the `setInterval()` method call, which uniquely identifies the interval method call. The patch involves clearing the timer just before the component is about to be destroyed i.e., in the component destructor[5].

[4]https://github.com/MTES-MCT/monitorfish/pull/953/commits/1dc01c0d82261bf05277366d954fa5d912632091

[5]https://github.com/MTES-MCT/monitorfish/pull/953/files

FP3b. Uncleared Interval Event

```
// I: Unique setInterval event ID
// D: Destructor method

- setInterval(() => {...})
+ I = setInterval(() => {...})
...
+ D() {
        ...
+       clearInterval(I)
+ }

// Example
{
   ...
  useEffect(() => {
-   setInterval(() => { setCount((prevCount) => {
      return prevCount - 1});
    }, 1000);

+   const intervalId = setInterval(() => {
     setCount((prevCount) => { return prevCount - 1});
    }, 1000);

+   return () => clearInterval(intervalId)
  }, []);
   ...
}
```

In the example snippet, setInterval is used within a useEffect hook (a React utility) to decrement a counter every second. However, this interval is never cleared, meaning the interval will continue to run even after the component is unmounted, leading to a memory leak.

The fix is similar to FP3a: The interval ID returned by setInterval is stored in a constant (`intervalId`). This allows us to reference this specific interval later. Now in React, the `useEffect` hook returns a cleanup function that is executed when the component is unmounted or before the effect is re-executed. Inside this cleanup function, `clearInterval` is called with the stored interval ID (intervalId). This stops the interval from continuing to run after the component is unmounted.

FP4. Uncancelled Animation Frame Requests. The `requestAnimationFrame()` Web API method helps determine the count of frames per second to allocate an animation, and execute the provided callback to perform that animation, before the actual screen loads [48]. Since it is used for creating animations on web pages, these are usually called recursively, which again leads to the risk of their execution post component destruction, retaining all objects in its callback function, even after they are no longer needed.

Fix: Similar to timers, each `requestAnimationFrame()` call also returns an ID unique to that specific request, that we can use to ensure the request is cancelled just before the component destroys[6].

[6]https://github.com/carbon-design-system/carbon-addons-iot-react/pull/2119/files

FP4. Uncancelled Animation Frame Request

```
// I: Unique requestAnimationFrame ID
// D: Destructor method

- requestAnimationFrame(() => {...})
+ I = requestAnimationFrame(() => {...})

+ D() {
        ...
+       cancelAnimationFrame(I)
+ }

// Example
{
   ...
-  requestAnimationFrame(this.animateSecondaryWaves);
+  this.frameId =
     requestAnimationFrame(this.animateSecondaryWaves);
   ...

+ componentWillUnmount() {
+   cancelAnimationFrame(this.frameId);
+ }
}
```

In the original version of the example code, `requestAnimationFrame` is used to schedule the custom `animateSecondaryWaves` method to be called before the next repaint. However, there is no reference stored to this request, and it is never canceled. This can lead to a memory leak if the animation continues to request new frames even after the component is unmounted.

To fix this, the request ID returned by `requestAnimationFrame` is stored in a class property (`this.frameId`). This allows the request to be referenced later. The `componentWillUnmount` lifecycle method is called just before the component is removed from the DOM. Inside this method, `cancelAnimationFrame` is called with the stored request ID (`this.frameId`). This cancels the scheduled animation frame request, ensuring that the animation does not continue to request new frames after the component is unmounted.

FP5a. Unremoved Component Instance Event Listeners. When the `$on` method is applied to a Vue component using `this.$on(event, callback)` [7,8] it means the component is listening for an event and will execute the provided callback function when this event is emitted. However when a component listens for events using `this.$on()`, it's important to clean up those listeners when the component is destroyed to prevent memory leaks.

Fix: Similar to previous fixes, the fix here is to remove the instance listener using the `$off()` method just before the component is destroyed[9].

[7]https://github.com/lan-ui/lan-ui/blob/a28f545e75dd8f444f7fdded965a27df9ac8dbe3/src/components/checkbox-group/checkbox-group.vue#L83

[8]https://github.com/ElemeFE/element/blob/290e68ea6aa6c56b7d83182b650e3be4f77ff1b0/packages/menu/src/menu.vue#L318

[9]https://github.com/nasa/openmct/pull/7070/commits/053f1a846c22427200e99a72fa13fac88e9a31ae#diff-501283c5c1b662d1c7a9e2215ca097c059991e1a0fe1d2736451be7d44c62747

FP5a. Unremoved Component Instance Event Listener

```
// E: Event
// C: Callback
// D: Destructor method

...
this.$on(E, C);
...
+ D() {
+     this.$off(E, C);
+ }

// Example
{
   ...
   mounted() {
      this.$on('submenu-click', this.handleSubmenuClick);
   },
   ...
+  beforeUnmount() {
+     this.$off('submenu-click', this.handleSubmenuClick);
+  }
}
```

FP5b. Unremoved Root Instance Event Listener

```
// E: Event
// C: Callback
// D: Destructor method

...
this.$root.$on(E, C);
...
+ D() {
+     this.$root.$off(E, C);
+ }

// Example
{
   ...
   mounted() {
     this.$root.$on('refetchPostComments', () => {
       this.refetchPostComments()
     })
   },
   ...
+  beforeUnmount () {
+     this.$root.$off('refetchPostComments')
+  }
}
```

In Vue, event listeners are managed within the component instance (this). In the original version of the example code, an event listener is added for the `submenu-click` event in the `mounted` lifecycle hook of Vue (`this.$on('submenu-click', this.handleSubmenuClick)`). This means that when the component is mounted, it starts listening for `submenu-click` events and executes the custom `handleSubmenuClick` method whenever the event is triggered.

Without proper cleanup, the event listener remains active even after the component is unmounted. This can lead to a memory leak because the event listener continues to hold a reference to the unmounted component, preventing it from being garbage collected.

The `beforeUnmount` lifecycle hook in Vue is called right before the component is unmounted from the DOM. The memory leak is fixed by calling `this.$off` with the event name (`submenu-click`) and the handler method, (`this.handleSubmenuClick`) in this hook, ensuring that the event listener is properly cleaned up.

FP5b. Unremoved Root Instance Event Listener. In Vue.js, `this.$root` refers to the root Vue instance, the top-level Vue component instance which is the parent of all other components. `this.$root.$on(event, callback)` is a method used to listen for custom events emitted from the root Vue instance[10,11]. Similar to the previous case, it will execute the provided callback function when the event is emitted. However, just like the regular Vue component, the root component also needs to clean up the listeners when it is destroyed, to prevent memory leaks.

Fix: The fix is similar to FP5a, i.e to remove the listener using the `$off()` method just before the root component is destroyed[12].

In the example code, `this.$root.$on('refetchPostComments', ...);` registers an event listener for the custom event 'refetchPostComments' on the root Vue instance (`$root`). The arrow function `() => this.refetchPostComments()` serves as the callback function that will be executed when 'refetchPostComments' event is emitted, during the `mounted()` lifecycle hook of the Vue component.

Again, if the event listener is not properly removed when the component is destroyed or unmounted, it can lead to memory leaks because JavaScript engine will keep holding references to the callback function `(() => this.refetchPostComments())` and associated data structures in memory.

To prevent this, in the `beforeUnmount()` hook, `this.root.off('refetchPostComments')` is called to remove the event listener for 'refetchPostComments' event from the root Vue instance (`$root`). This ensures that the callback function `(() => this.refetchPostComments();)` and associated resources are properly cleaned up when the component is about to be destroyed.

FP5c. Unremoved Event Bus Listener. In Vue.js, an event bus is a way to communicate and pass data between components that are not directly related or have a parent-child relationship. Event bus is created as a new Vue instance (`new Vue()`). Components can then emit events on the event bus, which can be listened to by any other component that is also using the same event bus [49]. However if a component subscribes to an event on the event bus but fails to unsubscribe when the component is destroyed, the event listener will continue to exist in memory, even though the component itself is no longer used. This can lead to a build-up of event listeners over time and eventually cause memory leaks.

Fix: The fix is similar to the above 2 cases, where the event needs to be removed from the event bus using the `$off()` method just before the root component is destroyed[13].

[10]https://github.com/lan-ui/lan-ui/blob/a28f545e75dd8f444f7fdded965a27df9ac8dbe3/src/components/checkbox-group/checkbox-group.vue#L83

[11]https://github.com/bootstrap-vue/bootstrap-vue/blob/5173dd19f6f46dc9d125cd7233fb59ccd2ef9296/docs/components/quick-links.vue#L55

[12]https://github.com/Ocelot-Social-Community/Ocelot-Social/commit/54ca9a6e0c9ebf7a39516622cb95f86c793176f5

[13]https://github.com/n8n-io/n8n/pull/6021/files#diff-323013d0d7d5d8ad10da80e95dd88d67aba4550d6bca4f64b3f95375adc710cf

TABLE I
DISTRIBUTION AND APPLICABILITY OF OUR FIX PATTERNS ACROSS THE THREE POPULAR SPA FRAMEWORKS

Fix Pattern	React	Angular	Vue
FP1	Applicable	N/A	N/A
FP2	Applicable	Applicable	Applicable
FP3	Applicable	Applicable	Applicable
FP4	Applicable	Applicable	Applicable
FP5	N/A	N/A	Applicable

FP5c. Unremoved Event Bus Listener

```
// B: Event bus
// E: Event
// C: Callback
// D: Destructor method

...
B.$on(E, C);
...
+ D() {
+      B.$off(E, C);
+ }

// Example
import { EventBus } from '../../services/EventBus'
...
{
    ...
    mounted() {
       EventBus.$on('refreshPeerList',
     this.debounceFetchPeers)
    },
    ...
+   beforeUnmount () {
+      EventBus.$off('refreshPeerList',
     this.debounceFetchPeers)
+   }
}
```

In the example, the `EventBus.$on('refreshPeerList', this.debounceFetchPeers)` is used to register an event listener (`debounceFetchPeers()`) that listens for the custom event 'refreshPeerList' emitted on the global EventBus.

By removing the listener from the bus using `EventBus.$off` in the `beforeUnmount()` hook, Vue ensures that resources associated with event handling function (this.debounceFetchPeers) are properly cleaned up when the component is about to be destroyed.

Our fix patterns are highly applicable to popular SPA frameworks as listed in Table I: React, Angular, and Vue. Notably, the majority of fixes are framework-agnostic. Even for those that are technically framework-specific (unsubscribing from subscriptions in Angular or removing event listeners from the EventBus in Vue), the underlying concepts remain consistent across frameworks. These fixes primarily address resource cleanup and memory management, making the approach broadly adaptable with minimal manual effort once the core principles are understood.

D. Edit Scripts for the Fix Patterns

For each individual fix pattern, we create a corresponding edit script to actually generate patches for potential memory leaks. An edit script is another program that parses the target program and locates potential leaking objects, where we apply the fix pattern. Each edit script has two components: (1) a potential leak object locator and (2) a patch writer. Creating edit scripts is a common procedure when applying a pattern-based program repair technique [16], [14], [50], [15], [35]. Therefore, we implement the scripts for our tool, which are available in our replication package [51].

Coverage of the Patterns

The 7 fix patterns cover most of the fixed memory leaks we have examined. Following the procedures described in Section III-B, we manually reviewed 124, 64 and 40 reported leaks in React, Angular and Vue applications respectively, that have been confirmed and fixed by the developers. Our fix patterns can non-intrusively fix 102 out of 124 (82%) leak types in React, 57 out of 65 (88%) already-known Angular-related memory leaks and 32 out of 40 (80%) known leaks in Vue. The full list of known memory leak bugs examined is available in our replication package [51].

E. Applying Fix Patterns

As the second step, LEAKPAIR applies the fix patterns extracted in the first step (Section III-B). Basically, we assume that one can apply LEAKPAIR to the whole project by scanning the source code tree of the project, which implies that the edit scripts explained in Section III-D are executed for each file. Specifically, it follows the following procedure.

- **Parsing and Detecting:** LEAKPAIR makes use of the Babel compiler [52] in conjunction with Facebook's *jscodeshift* [53] to traverse through the JS file (in the case of a single file path) or all JavaScript files from the root of the given project path. For each file, it extracts the AST by leveraging the Babel compiler. During the AST traversal, LEAKPAIR detects Angular, React, and Vue components by matching their syntax definition. Once a component from these frameworks is identified, it detects whether the component implements any of the five memory leak patterns by traversing the AST, visiting each node, and matching the patterns illustrated in Section III-C.
- **Creating Patches:** If a leak pattern is matched, it tracks the file name as well as how many objects are leaking due to that leak type, i.e., are following the same pattern, in that specific component. It then generates and adds the fix in the AST. After the patch is successfully applied, it updates the count of potentially leaking objects for that leak type, in the project/file. Finally, it then converts the AST back to source code by leveraging the Recast [54] library.
- **Repeating and Reporting:** LEAKPAIR repeats this process for all the files if a project path was specified; otherwise, the processing completes there. At the end of the execution, it prints out the repaired file name(s) and the total count of each leak type in the console (from which the LEAKPAIR command was executed) as well as in an external `json` file (if an output path was specified in the command).

F. Non-Intrusive Patch Generation

To ensure that LEAKPAIR generates non-intrusive patches for memory leaks in SPAs, we enforce the following conditions in

addition to the steps of standard pattern-based program repair techniques [16], [15]:

- **Localizing without test cases:** Since LEAKPAIR proactively generates patches for memory leaks in SPAs, it does not rely on external fault localization techniques usually based on test suites. Instead, our approach scans specific objects in the source code. For example, FP1 detects all `Observable` objects in the target SPA.
- **Avoiding redundant fix:** Among the detected target objects, some of them are correctly used and memory leaks are prevented, where LEAKPAIR does not need to generate corresponding patches, for example, a subscription to an observable that is already released. LEAKPAIR remains idempotent by leaving such code unchanged. LEAKPAIR achieves this through a simplistic AST parsing approach, traversing the AST to identify nodes and relationships matching leaky patterns and applying fixes only where necessary. For example, if an event listener is added but not removed in the destructor, LEAKPAIR will apply the fix. However, if it detects that the corresponding `removeEventListener` is already called with the correct target and listener nodes, it will leave the code as is.
- **Checking non-intrusiveness:** For each generated patch, LEAKPAIR examines whether the patch breaks any functionality. As the regression test suites are often available for a target SPA, our approach runs the suites to find any behavior changes. Although test cases may not guarantee complete behavior integrity, the test results may show the correctness of key functionalities for the target SPA.

IV. EVALUATION

A. Research Questions

Our experiments investigate the following research questions:

1) **RQ1. (Effectiveness)** How effective is the tool at minimizing/eliminating memory leaks?
2) **RQ2. (Acceptability)** How useful are generated patches, as perceived by developers?
3) **RQ3. (Non-intrusiveness)** What is the impact of our tool on test suite execution results?
4) **RQ4. (Comparison against GPT-4)** How does LEAKPAIR compare to GPT-4 in terms of their performance?

The first research question is designed to assess the amount of memory reduction when applying LEAKPAIR to SPAs. For this RQ, we collect a set of known memory leaks and another set of unknown leaks in open-source projects as experiment subjects. We apply our tool to the subjects and examine their memory footprints before and after repair.

While RQ1 assesses the effectiveness, RQ2 focuses on whether the patches generated by LEAKPAIR can be accepted by the developers of the open-source projects. As the unknown leaks used in the experiments for RQ1 are in fact new defects, we report them as new pull requests and see whether they are merged or accepted.

As LEAKPAIR is designed to generate non-intrusive patches, it is necessary to assess whether the patches disrupt the functionality of the target subjects or cause compilation errors. Therefore, we designed RQ3 to assess non-intrusiveness. Our experiments for this RQ try to compile the subject programs used in the previous RQs and run the test cases already given for the programs.

Finally, RQ4 is to compare LEAKPAIR with GPT. Recent studies have shown the effectiveness of GPT in fixing functionality bugs, and in this work, we compare the effectiveness of GPT in fixing memory leaks of SPAs with LEAKPAIR. To the best of our knowledge, this is the first study to compare the effectiveness of GPT in fixing memory leaks of SPAs. In this study, we use GPT-4, which is the latest version of GPT at the time of conducting the experiments.

B. Experiment Setup

We used the following experiment design to answer the research questions described in Section IV-A.

1) Subjects: To assess the effectiveness of our tool, we collected SPAs based on the following criteria:

- **Maintained.** We choose projects that are still being maintained and whose last update was less than a year ago. Archived projects are not considered.
- **Number of contributors.** Projects with at least 10 contributors are selected. Personal projects are not taken into account.
- **Number of commits.** The selected projects have at least 100 commits on their GitHub repository.
- **Popularity.** Projects with at least 10 stargazers, watchers, or forks are selected.
- **Framework.** The selected projects should use either React or Angular as their base framework, as our target is SPAs.

Based on the above criteria, we collect a set of projects with *unknown* new memory leaks and another set of projects with already *known* leaks (i.e., those fixed by the developers). The projects with already known leaks are necessary to show whether our tool can reproduce the patches generated by the developers of the projects. Other projects are collected to assess the effectiveness of our tool in discovering and repairing new and unknown memory leak patterns.

As a result, 60 projects are selected as the subjects for our experiments to assess LEAKPAIR ; 30 projects have unknown new memory leaks while 30 projects out of them have already known memory leaks. Tables II and III list the 30 unknown and 30 known subjects, respectively.

2) Repairing Memory Leaks: To answer RQ1, our first experiment applies our tool to the subjects described in Section IV-B1. We run LEAKPAIR on the root of each subject so that it scans the project directories and identifies JavaScript files. For each source code file, the tool tries to change the file by applying each pattern. Our tool addresses all locations if applicable.

After applying LEAKPAIR, we then measure the memory footprints. Because we need to run the target subject to determine memory consumption, we create a scenario file for each subject. Using scenario files is a common procedure when

measuring the memory consumption of web applications. For example, BLeak [12] and MemLab [44], the most recent techniques to detect memory leaks, require scenario files to run the target web applications. The scenario files used for each subject are available in our replication package [51].

To compare the memory consumption, we compute the memory footprints before and after applying LeakPair. For each subject, the corresponding scenario file is executed 25 times with `loop=10` (i.e., 25 × 10 times in total for each subject) since a single loop may not accurately reveal the memory consumption. We then collect memory consumption in megabytes (MB) and the number of object clusters [44], where a cluster is the collection of all retainer paths for all the leaking objects due to a single leak origin. Applying the Mann-Whitney U test [108], we compute the statistical significance of the differences between values before and after patches. Note that this is not a stage of LeakPair ; rather, this is only for the evaluation.

3) Reporting Generated Patches: As the unknown memory leaks are basically newly found bugs, we report the leaks to the repositories of the subjects. For each patch generated by our tool, we create a pull request with the patch and memory footprints before and after applying the tool. The outcome of the reported pull requests can be `Agreed`, `Disagreed`, or `Ignored`. The 3 types of outcomes for our PRs are recorded to answer RQ2.

4) Running Test Cases on Patches: To figure out whether the patches generated by LeakPair break the functionality of the subjects, we execute the test cases available in the subjects and count the number of passing and failing cases. As most of the popular open-source projects maintain (regression) test suites, we simply run the test cases included in the subjects. Many subjects use test automation frameworks; in that case, we resort to those frameworks; otherwise, we follow the instructions available in the contribution guide for each subject. We also compare the number of passing/failing test cases before and after applying LeakPair. The results of this experiment can answer RQ3.

5) Repairing Memory Leaks With GPT: In this study, we primarily focus on evaluating the patch generation capability of GPT-4, while assuming a separate process for fault localization. That is, we directly pass a code snippet causing a memory leak to GPT and ask it to fix the leak. This approach is similar to the *perfect fault localization* assumption widely adopted in the APR literature [15], [22]. Under the perfect fault localization assumption, APR tools are provided with the exact fault location and generate a patch to fix it. The perfect fault localization assumption is used under the following APR scenario. First, a ranked list of suspicious locations is generated by using a fault localization technique. Then, the APR tool iterates over this list and generates patches for each suspected location. This approach works on the premise that a fault can be confined to a single location, whether it be a single line, a single function, or a single file.

We use the above perfect fault localization assumption because it is difficult to perform fault localization with GPT due to the limited size of the context window—GPT-4's context window size is bound to 8,192 tokens. Assuming perfect fault localization, we focus on the patch generation capability of GPT-4. In comparison, recall that LeakPair performs fault localization and patch generation simultaneously by scanning the AST of each source code file in a project; the size of the program is generally not a concern for LeakPair. Due to the GPT's context-window size limitation, we provide a more favorable condition for GPT compared to LeakPair by assuming perfect fault localization.

TABLE II
Subjects With Unknown Memory Leaks

ID	Program	Type	SPA Framework	Commit Hash
U1	react-zoom-pan-pinch [55]	Library	React	`fdc030`
U2	Angular Extentions Elements [56]	Library	Angular	`d9a4e4`
U3	Evergreen [57]	Framework	React	`82c3a8`
U4	ngx-datatable [58]	Library	Angular	`6184c9`
U5	react-multi-carousel [59]	Library	React	`525793`
U6	codetekt (Frontend) [60]	Website	Angular	`7b8289`
U7	skbkontur/retail-ui [61]	Framework	React	`32f3cf`
U8	Aam Digital [62]	Web app	Angular	`304ff9`
U9	Replay's DevTools [63]	Library	React	`24d10f`
U10	ngx-bootstrap [64]	Framework	Angular	`663c70`
U11	DefichainIncome [65]	Web app	Angular	`911509`
U12	Collosal [66]	Web app	React	`798e7a`
U13	The Book Thieves [67]	Web app	React	`6de1a6`
U14	Mempool [68]	Web app	Angular	`5905ee`
U15	DSpace [69]	Web app	Angular	`b29dd6`
U16	PrimeNG [70]	Library	Angular	`085a4e`
U17	Formly [71]	Library	React	`e262fb`
U18	Foxglove Studio [72]	Web app	React	`d49827`
U19	BootstrapVue [73]	Framework	Vue	`5173dd`
U20	Chatwoot [74]	Framework	Vue	`11b27f`
U21	think-vuele [75]	Library	Vue	`2e256f`
U22	vue-admin-better [76]	Framework	Vue	`f34069`
U23	Vue Grid Layout [77]	Library	Vue	`6e5367`
U24	Weaverbird [78]	Framework	Vue	`1aa799`
U25	AutoAnimate [79]	Library	Vue	`b62aa8`
U26	vue-snap [80]	Library	Vue	`2cb14b`
U27	Element [81]	Library	Vue	`290e68`
U28	lan-ui [82]	Framework	Vue	`a28f54`
U29	iView [83]	Library	Vue	`c3f57f`
U30	Buefy [88]	Library	Vue	`b469a3`

We reuse the 30 projects with known memory leaks (see Table III) used in RQ1. Recall that for those projects, developer-written patches for memory leaks are available. We apply GPT to the program location modified by the developers, as described in the previous paragraph. Meanwhile, we do not reuse the unknown subjects listed in Table II for this experiment, due to the lack of developer-written patches.

Our experiment for this RQ asks GPT to generate patches with the following procedure. The faults in our subjects are confined to a single location with diverse granularity. In some subjects, the fault is confined to a single function (i.e., the developer-written patch modifies a single function), while in others, it is confined to a single class or a single file. Depending on the granularity of the developer-written patch, we pass the corresponding code snippet to GPT. For instance, if the developer-written patch is confined to a single function, we pass the buggy function to GPT. If the developer-written patch is confined to a single class, we pass the buggy class to GPT. In eight subjects out of 30, the developer-written patches modify multiple files. In seven out of these eight subjects, the developer-written patches consist of multiple atomic patches, where each atomic patch is independent of the others and is confined to a granularity smaller than a file. For these seven subjects, we treat

TABLE III
SUBJECTS WITH KNOWN MEMORY LEAKS

ID	Program	Type	SPA Framework	Commit Hash
K1	react-zoom-pan-pinch [55]	Library	React	6e35b3
K2	Fundamental Library for Angular [85]	Library	Angular	be9629
K3	react-multi-carousel [59]	Library	React	5d252d
K4	Angular Components [56]	Framework	Angular	1bbb29
K5	Material UI [86]	Framework	React	e92b1c
K6	Angular Components documentation [87]	Website	Angular	e8cb0d
K7	Rooster [6]	Library	React	c3f2f0
K8	Octant [88]	Framework	Angular	b079ad
K9	Evergreen [57]	Framework	React	a716f4
K10	Transloco [89]	Library	Angular	2338a0
K11	MonitorFish [90]	Website	React	6e35b3
K12	react-customizable-progressbar [91]	Library	React	4b0af1
K13	Secret Network Oracle Client [92]	Library	React	5d252d
K14	Mempool [68]	Website	Angular	631de8
K15	Momentum Mod [93]	Website	Angular	203707
K16	PatternFly React [94]	Library	React	f4d651
K17	React Number Format [95]	Library	React	11de23
K18	Help Scout Design System [96]	Library	React	b079ad
K19	mappit [97]	Website	React	be6979
K20	Vue-Tree [98]	Library	Vue	c53d34
K21	Open MCT [99]	Web app	Vue	cd5699
K22	clr.fund [100]	Web app	Vue	8184bc
K23	web-mapviewer [101]	Web app	Vue	e97f4c
K24	InstaLog [102]	Web app	Vue	89bba5
K25	n8n [103]	Web app	Vue	053a4f
K26	Element [81]	Library	Vue	23e818
K27	Prefect UI [104]	Framework	Vue	6e889b
K28	PyCon TW official website [105]	Website	Vue	268e12
K29	2N.fm [106]	Web app	Vue	891d9a
K30	Documentation for Vue 3 [107]	Website	Vue	5b111e

A JavaScript function provided below may contain memory leak(s). Please, fix all memory leaks that you can find.

1. If you cannot find any memory leaks, just reply "no leaks detected".
2. If you found a memory leak, modify the provided JavaScript function and fix the memory leak.
3. You should return the entire function that I provided after fixing the memory leaks.
4. Do not include comments like "// Rest of your component here....", "/ rest of JSX class here */", or "/* rest of the component remains unchanged */" because you SHOULD return the fixed function entirely.*
5. Surround the fixed function with triple backticks (```).
6. Do not include any explanations.

JavaScript function:

```
[the function subject to repair]
```

Fig. 5. Prompt template for function-level granularity. "[the function subject to repair]" is replaced with the code snippet. The templates for the other granularities are similar.

each modified file separately and decide the granularity based on the developer-written patch for that file. In the remaining subject (i.e., K20), an atomic patch encompasses multiple files and we exclude this subject from our evaluation.

For each file subject to repair, we collect 10 patches using GPT-4 by running the prompt 10 times. Fig. 5 shows the prompt template we use for the function-level granularity. We use similar prompts for class- and file-level granularities.

We consider a patch *plausible* if it satisfies the following conditions: (1) the subject runs without any errors when the patch is applied, and (2) there is no regression error. For each plausible patch, we measure the memory consumption using MemLab before and after applying the patch. We consider a patch *valid* if it reduces the memory consumption.

V. RESULTS

This section presents and analyzes the results of experiments to answer the research questions described in Section IV.

A. RQ1: How Effective Is LeakPair?

The patches generated by LEAKPAIR can reduce memory consumption, as shown in Tables IV and V. We apply the tool to each subject listed in Table II (projects with unknown memory leaks) and III (projects with already known leaks) according to the procedure described in Section IV-B2. In the result tables, the `Leak Patterns` column lists the fix patterns (see Section III-C) successfully applied to each subject. The `Leaked Objects *` columns represent the number of clusters in which objects are potentially leaking the memory space, before and after applying our tool, and the difference. The `Heap Size *` columns show the average heap size based on the 25 iterations before and after applying our tool, and the difference.

As shown in Tables IV and V, respectively, LEAKPAIR reduces memory consumption in the majority of the subjects. The statistical significance of the differences are denoted as $*$:p-value$<$0.05 and $**$:p-value$<$0.01.

The reduction is relatively larger for the subjects with unknown leaks. This seems to be because the subjects with known leaks tend to be better maintained in terms of memory management than the subjects with unknown leaks. Note that for the subjects with known leaks, we reverted the patches applied by the developers to introduce the leaks. The developers for these subjects are aware of the leaks and are likely to pay more attention to memory management, which could explain the smaller reduction in memory consumption.

In contrast to the general pattern, memory consumption increases in subject **U25** after applying LEAKPAIR, although the increase is only 0.9 MB. However, the number of leaked clusters decreased from 11 to 2, indicating that the applied fix is effective. A possible explanation for the increase in memory consumption despite the reduction in the number of leaked clusters is that while the fix eliminated 9 leaked clusters out of 11, the remaining two clusters may contain garbage objects that are not collected by the garbage collector yet. Note that MemLab, the tool we used to measure memory consumption, can be conservative when detecting leaked objects. Although garbage objects will be eventually collected, they may temporarily increase memory consumption until the garbage collector runs.

The plots in Figs. 6 and 7 illustrate the changes in the memory heap size before and after the leak fixes. The horizontal axis represents the 25 iterations, while the vertical axis denotes the minimum to maximum range of the heap size of each subject (with or without a fix). `no Patch`, represented with pink lines, denotes the heap size *before* the fix, while the blue line is for heap size *after* applying the patch by LEAKPAIR.

Although there was some fluctuation due to the nature of web applications (e.g., it can be affected by the browser status even for the same scenarios), it turns out that our patches contribute

TABLE IV
MEMORY CONSUMPTION RESULTS BEFORE AND AFTER APPLYING LEAKPAIR TO THE SUBJECTS IN TABLE II

ID	Leak Patterns	Leaked Objects Before applying LEAKPAIR	Leaked Objects After applying LEAKPAIR	Leak Object Reduction	Heap Size Before applying LEAKPAIR	Heap Size After applying LEAKPAIR	Total Heap Size Reduction
U1	FP3	5.5 clusters	5 clusters	0.5 cluster	31.9 MB	31.5 MB	0.4 MB (1.3%)∗
U2	FP1, FP2	9 clusters	8.5 clusters	0.5 clusters	17.3 MB	16.7 MB	0.6 MB (3.5%)∗∗
U3	FP4	5 clusters	3 clusters	2 clusters	34.3 MB	29.2 MB	5.1 MB (14.9%)∗∗
U4	FP3, FP4	8 clusters	7 clusters	1 cluster	367.1 MB	364.9 MB	2.2 MB (0.6%)
U5	FP3	4 clusters	3.5 clusters	0.5 cluster	20.9 MB	20.9 MB	0 MB (0%)∗
U6	FP1, FP3	10.5 clusters	9 clusters	1.5 clusters	44.2 MB	43.5 MB	0.7 MB (1.6%)
U7	FP3	6.5 clusters	5.5 clusters	1 clusters	229.2 MB	222.8 MB	6.3 MB (2.7%)∗
U8	FP1	2 clusters	2 clusters	0 clusters	322.5 MB	265.8 MB	56.7 MB (17.6%)∗∗
U9	FP3	5 clusters	3 clusters	2 clusters	27 MB	26.4 MB	0.6 MB (2.2%)
U10	FP1	6.5 clusters	6 clusters	0.5 clusters	101.4 MB	101 MB	0.4 MB (0.3%)
U11	FP1	5 clusters	4 clusters	1 cluster	60.5 MB	60.4 MB	0.1 MB (0.1%)
U12	FP2	6 clusters	5 clusters	1 cluster	83.7 MB	40.7 MB	43 MB (51.4%)
U13	FP3	2 clusters	1 cluster	1 cluster	56.9 MB	54.3 MB	2.5 MB (4.4%)
U14	FP1, FP4	2 clusters	1 cluster	1 cluster	88.9 MB	63.6 MB	25.3 MB (28.3%)∗∗
U15	FP1	4.5 clusters	4.5 clusters	0 cluster	69.05 MB	68.65 MB	0.4 MB (0.6%)∗∗
U16	FP1	6.5 clusters	5.5 clusters	1 cluster	20.2 MB	19.5 MB	0.7 MB (3.5%)
U17	FP1, FP2	2.5 clusters	2.5 clusters	0 cluster	13.8 MB	13.5 MB	0.3 MB (2.5%)
U18	FP2, FP3	6 clusters	6 clusters	0 clusters	61.8 MB	61.8 MB	0 MB (0%)
U19	FP5b	7 clusters	4 clusters	3 clusters	198 MB	163.6 MB	34.4 MB (17.4%)
U20	FP2, FP5c	0 clusters	0 clusters	0 clusters	295.2 MB	281.3 MB	13.9 MB (4.7%)
U21	FP5c	2 clusters	1 clusters	1 clusters	48.8 MB	39.6 MB	9.2 MB (18.9%)
U22	FP2, FP5c	11 clusters	11 clusters	0 clusters	181.7 MB	176.4 MB	5.3 MB (3%)*
U23	FP2	0 clusters	0 clusters	0 clusters	16.7 MB	16.6 MB	0.1 MB (0.6%)*
U24	FP2	6.5 clusters	6 clusters	0.5 clusters	216.9 MB	213.6 MB	3.3 MB (1.5%)
U25	FP2, FP3	11 clusters	2 clusters	9 clusters	12.6 MB	13.5 MB	0 MB (0%)**
U26	FP5a	2.5 clusters	2.5 clusters	0 clusters	21.1 MB	20.8 MB	0.4 MB (2.3%)
U27	FP2, FP5c	3 clusters	3 clusters	0 clusters	65.7 MB	63 MB	2.7 MB (4.1%)
U28	FP2, FP5c	0 clusters	0 clusters	0 clusters	24.4 MB	24.4 MB	0 MB (0%)
U29	FP2, FP5a	7 clusters	7 clusters	0 clusters	161.5 MB	158.9 MB	2.6 MB (1.6%)**
U30	FP2, FP5c	11.4 clusters	10.6 clusters	0.8 clusters	401.6 MB	401.6 MB	0 MB (0%)

∗: p-value $<$ 0.05, ∗∗: p-value $<$ 0.01.

TABLE V
MEMORY CONSUMPTION RESULTS BEFORE AND AFTER APPLYING LEAKPAIR TO THE SUBJECTS IN TABLE III

ID	Leak Patterns	Leaked Objects Before applying LEAKPAIR	Leaked Objects After applying LEAKPAIR	Leak Object Reduction	Heap Size Before applying LEAKPAIR	Heap Size After applying LEAKPAIR	Total Heap Size Reduction
K1	FP2	4.5 clusters	4 clusters	0.5 clusters	32 MB	32 MB	0 MB (0%)
K2	FP1	0 cluster	0 cluster	0 clusters	55.6 MB	55.5 MB	0.1 MB (0.2%)∗
K3	FP3	3 clusters	2.2 cluster	0.8 cluster	25.9 MB	24.9 MB	1 MB (0%)∗
K4	FP1	10 clusters	10 cluster	0 cluster	56.9 MB	44.6 MB	12.3 MB (21.6%)∗∗
K5	FP3	10.4 clusters	10 clusters	0.4 clusters	15.05 MB	15.05 MB	0 MB (0%)
K6	FP1	1 clusters	0.6 clusters	0.4 clusters	15.05 MB	15.05 MB	0.0 MB (0%)
K7	FP2	3 clusters	3 clusters	0 clusters	17.45 MB	17.45 MB	0 MB (0%)
K8	FP1	13 clusters	12.5 clusters	0.5 clusters	161.6 MB	161.3 MB	0.3 MB (0.2%)
K9	FP3	4.5 clusters	4.5 clusters	0 clusters	33.8 MB	33 MB	0.8 MB (2.4%)∗
K10	FP1	0.5 cluster	0.5 cluster	0 cluster	9.2 MB	9.2 MB	0.0 MB (0%)
K11	FP2, FP3	1 clusters	1 clusters	0 clusters	66.2 MB	66.1 MB	0.1 MB (0.2%)
K12	FP3	1 cluster	1 cluster	0 cluster	7.2 MB	7.1 MB	0.1 MB (1.4%)
K13	FP2	2.5 clusters	2.4 cluster	0.1 cluster	86.2 MB	86.2 MB	0 MB (0%)∗
K14	FP2	7 clusters	7 clusters	0 cluster	89 MB	88.9 MB	0.1 MB (0.1%)
K15	FP1	0 clusters	0 clusters	0 clusters	31.2 MB	31.2 MB	0 MB (0%)
K16	FP3	2.5 clusters	2.5 clusters	0 clusters	6 MB	5.8 MB	0.2 MB (3.3%)
K17	FP3	0 clusters	0 clusters	clusters	7 MB	7 MB	0 MB (0%)
K18	FP3	1.3 clusters	1.1 clusters	0.2 clusters	34.5 MB	34.5 MB	0 MB (0%)
K19	FP1	3.5 clusters	2 clusters	1.5 clusters	20.3 MB	20.1 MB	0.2 MB (0.1%)∗∗
K20	FP5c	0 clusters	0 clusters	0 clusters	12.1 MB	12.1 MB	0 MB (0%)
K21	FP5a	10 clusters	10 clusters	0 clusters	269.2 MB	269.1 MB	0.1 MB (0.03%)
K22	FP3	4 clusters	3.5 clusters	0.5 clusters	57.7 MB	57.2 MB	0.5 MB (0.9%)
K23	FP2, FP3	1 clusters	1 clusters	0 clusters	113.8 MB	113.5 MB	0.3 MB (0.3%)∗
K24	FP3	0 clusters	0 clusters	0 clusters	9.2 MB	9.1 MB	0.1 MB (1.1%)
K25	FP2	13 clusters	12.9 clusters	0.1 clusters	209 MB	209.6 MB	0 MB (0%)
K26	FP3	1 clusters	1 clusters	1 clusters	52 MB	52 MB	0 MB (0%)
K27	FP2	26 clusters	25 clusters	1 clusters	150.8 MB	149.2 MB	1.6 MB (1.1%)
K28	FP2	1 clusters	0.8 clusters	0.2 clusters	40.5 MB	40.5 MB	0 MB (0%)
K29	FP2	0 clusters	0 clusters	0 clusters	8.3 MB	8.3 MB	0 MB (0%)
K30	FP2	10.5 clusters	10.5 clusters	0 clusters	18.1 MB	15.3 MB	2.8 MB (45.3%)

∗: p-value $<$ 0.05, ∗∗: p-value $<$ 0.01.

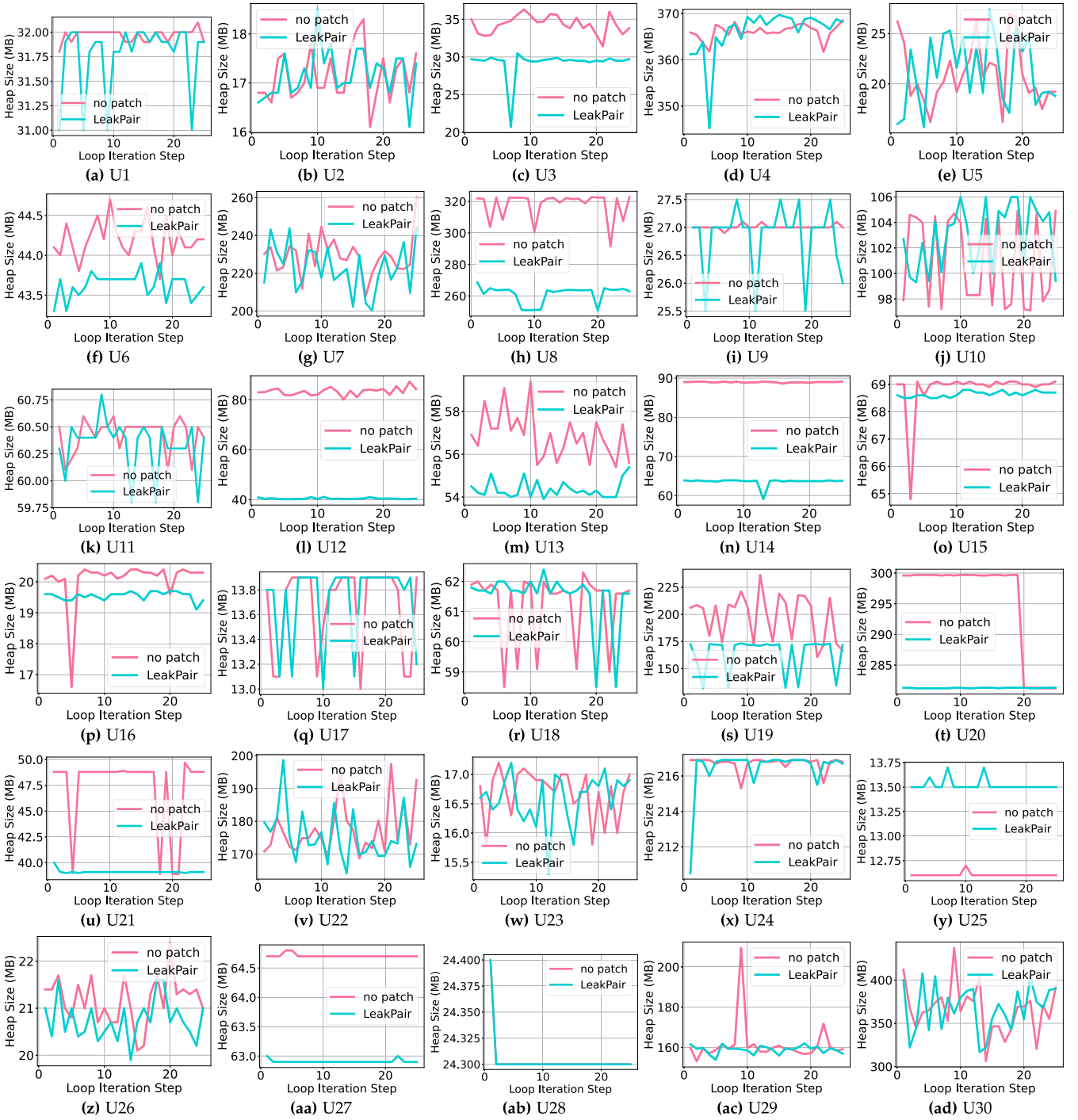

Fig. 6. Heap size over loops after applying LeakPair to the subjects listed in Table II.

to reducing memory consumption, or at least, they do not add to it, nor do they introduce any new leaks. Again, it is noticeable how the subjects with unknown leaks (Fig. 6) showed a significant heap reduction as compared to the ones with known leaks (Fig. 7).

The results of our experiments may imply that LEAKPAIR is effective for most SPAs, no matter how it is maintained. It might be helpful to reduce memory consumption, and it can further prevent potential memory bloats. Furthermore, it does not add any harmful code and does not increase memory consumption in any way.

Answer to RQ1: LEAKPAIR *can generate patches to fix memory leaks in SPAs without leak detection, and the patches successfully reduce applications' memory consumption. It turns out that they are competitive with the original patches written by human developers.*

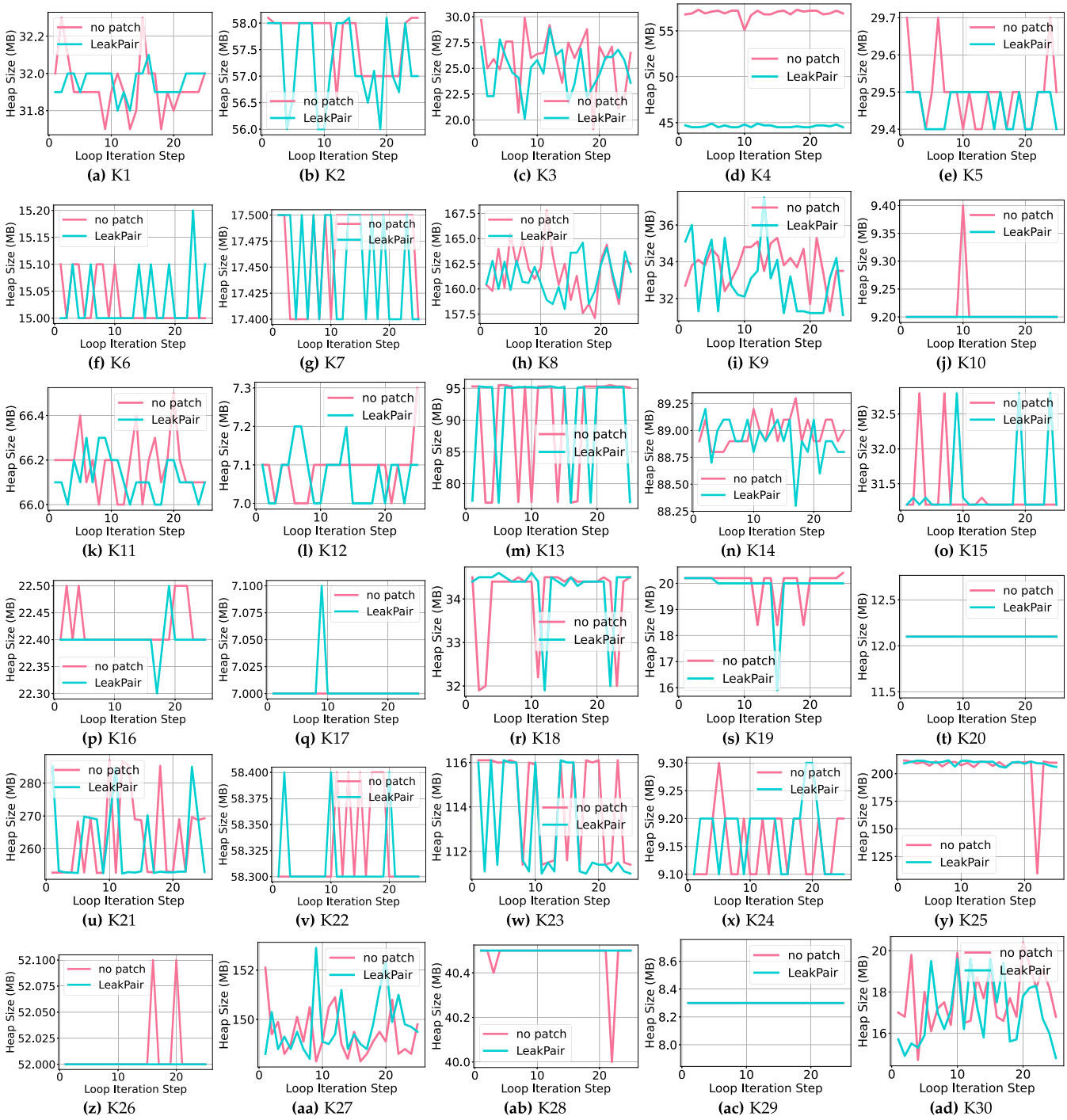

Fig. 7. Heap size over loops after applying LeakPair to the subjects listed in Table III.

B. RQ2: Are the Patches by LeakPair Acceptable?

To assess the acceptability of patches generated by LeakPair, a live study was carried out on active open-source SPA projects (including SPA websites and libraries used by them), as described in Section IV-B3.

The study involves creating pull requests (PRs) for patches by LeakPair for the subjects in Table II, and observing the outcome of pull requests. We submitted 32 pull requests after clustering similar leaks/patches and confirming a substantial reduction in the count of memory leaks or heap size by the patches, together with the analysis results by Memlab [44].

Table VI contains the results of the live study up to the submission date. `Merged` refers to PRs that were directly merged in the original form. `Improved` represents the cases where the developer(s) do not merge the original PR, but rather create a new PR to address the issues or make improvements in their

TABLE VI
RESULTS OF PULL REQUESTS REPORTING THE PATCHES GENERATED BY LEAKPAIR, WHICH FIX UNKNOWN LEAKS IN SUBJECTS LISTED IN TABLE II

Agreed			Disagreed	Ignored	Total
Merged	Approved	Improved			
14	2	0	0	16	32

code themselves, based on our PR submission. Sometimes the project authors acknowledge the issues that were fixed by the PR but the merging of the PR remains pending or undecided, the `Approved` section covers these cases. `Disagreed` denotes the instances where the developer rejects acknowledging the issue entirely, while `Ignored` are PR that received no action (tag/comment/reaction) by the project authors up till the date of submission.

Sixteen out of 32 PRs (50%) are approved by the developers, out of which 9 were merged directly. One PR led to the creation of a separate PR by the project developers based on the changes in our PR, which addressed the same leak patterns but used a slightly different approach (in compliance with their specific programming conventions), which was then merged. The leak patterns repaired in 2 of the PRs are approved as anti-patterns by the authors that need to be addressed; however, the PRs for them have not yet been merged. The authors have taken note of our repairs and plan to address the leak patterns themselves soon.

One of our PRs inspired the project owner to fix a similar memory leak pattern as the one in the PR. It is worth noting that no PR has been rejected so far, which further corroborates the non-intrusive nature of LEAKPAIR patches. 16 PRs did not get any response from the developers up to the date of submission.

Answer to RQ2: *The patches generated by* LEAKPAIR *are even acceptable to the developers of the target projects. While more than half of the patch suggestions are accepted, there are no explicitly rejected patches.*

C. RQ3: Do the patches break the functionality?

Fortunately, our approach of ensuring correctness benefits from the modular structure of the single-page applications, where each component is typically written in a separate file (module). Even if a specific test case for a particular component is not available, the shared nature of code among modules means that a failure in one component is likely to impact others, and at the very least, the root component. When this happens, the application would most likely fail to compile successfully or break at least some test cases.

To show the non-intrusiveness of the patches generated by our tool, we built and compiled the application after the patch application by LEAKPAIR. Then we ran the test cases of each subject according to the procedure explained in Section IV-B4. We could not run test suites for two and four subjects listed in Tables II and III, respectively. Across the 40 projects containing test cases, we evaluated a total of 15,315 test cases, with an average of 382.88 test cases per project. The number of test cases per project ranged from 1 to 4,272. The tables report on the execution time of the test suites as well.

As shown in Tables VII and VIII, the patches generated by LEAKPAIR do not introduce any new positive or negative test outcomes 99% of the time. For one subject (U28), we see new failing test cases after LEAKPAIR fixes; this is because one of the fix patterns (Event listener leak fix) used by LEAKPAIR makes use of a web API (AbortController [109]) which is supported by all modern mainstream browsers; however, the dynamic tests provided within the project U28 were written using an obsolete browser that does not support the API.

For subjects with some skipped and failed test cases in the original version, we checked if any new positive or negative test cases had replaced the previous outcomes. Again, we found no discrepancies (other than for U28), indicating that our patches do not change the behaviors of the subjects, at least with respect to the test suites provided. In addition, no significant differences were noted with respect to test execution times either, as can be seen from the columns *Elapsed time before applying* LEAKPAIR and *Elapsed time after applying* LEAKPAIR.

The results of this experiment show that LEAKPAIR is unlikely to break the functionality of SPAs when generating patches to fix potential memory leaks. This implies that the users of LEAKPAIR may apply the tool without having the functionality changed. Although running test suites may not guarantee the non-intrusiveness of patches, our tool is highly likely to generate patches that preserve the behaviors of the programs.

Answer to RQ3: *According to the test results, the patches by* LEAKPAIR *are not intrusive. Although test suites cannot guarantee their correctness, the patches do not break any functionality, at least from a maintenance perspective.*

D. RQ4: How Does LeakPair Compare With GPT-4?

Table IX shows the results for RQ4. The first column shows the project IDs and the second column shows the heap size measured by MemLab when the original buggy project is used. The third column shows the file IDs; each file modified in the developer-provided patch is assigned a unique ID. The fourth column shows the patch granularity used to modify the file using GPT-4, which can be one of the following: function, class, or file. For each file, we retrieve ten patches generated by GPT-4. The number of plausible patches and valid patches are shown in the seventh and eighth columns, respectively. Occasionally, the generated patches fail to run the target SPA project and the "Run Failure" columns show the number of such patches. Even if the patches run successfully, they may not pass the test suite, and the "Test Fail" column shows the number of patches that fail to pass the test suite.

We also found that MemLab fails to run for 37 patches as shown in the "MemLab Failure" column. The following is the breakdown of the failures: 34 times due to the time out error; the tested scenario cannot retrieve a UI (User Interface) element within the time limit, and 3 times due to the other errors caused by the mismatch between the scenario file and the actual web

TABLE VII
TEST EXECUTION APPLYING LEAKPAIR TO THE SUBJECTS IN TABLE II

ID	Test results before applying LEAKPAIR	Test results after applying LEAKPAIR	Elapsed time before applying LEAKPAIR	Elapsed time after applying LEAKPAIR
U1	N/A	N/A	N/A	N/A
U2	N/A	N/A	N/A	N/A
U3	46 passed of 46	46 passed of 46	8.1 s	8.4 s
U4	126 passed of 129	126 passed of 129	0.3 s	0.3 s
U5	6 passed of 14	6 passed of 14	3.9 s	1.4 s
U6	101 passed of 101	101 passed of 101	55 s	56.6 s
U7	66 passed of 66	66 passed of 66	119.835 s	120.835 s
U8	1031 passed of 1038	1031 passed of 1038	41.6 s	43.7 s
U9	43 passed of 43	43 passed of 43	6.5 s	6.3 s
U10	12 passed of 12	12 passed of 12	0.3 s	0.3 s
U11	N/A	N/A	N/A	N/A
U12	N/A	N/A	N/A	N/A
U13	N/A	N/A	N/A	N/A
U14	N/A	N/A	N/A	N/A
U15	4272 passed of 4272	4272 passed of 4272	47.3 s	1 min 18.4 s
U16	788 passed of 788	788 passed of 788	38.8 s	39.8 s
U17	56 passed of 56	56 passed of 56	9 s	8.8 s
U18	120 passed of 120	120 passed of 120	42 s	43.3 s
U19	164 passed of 164	164 passed of 164	11.5 s	8.6 s
U20	239 passed, 14 failed of 253	239 passed, 14 failed of 253	46.4 s	41.6 s
U21	N/A	N/A	N/A	N/A
U22	N/A	N/A	N/A	N/A
U23	18 passed of 18	18 passed of 18	10.6 s	2.3 s
U24	1961 passed, 9 skipped of 1970	1961 passed, 9 skipped of 1970	28.5 s	27.7 s
U25	5 failed, 1 passed of 6	5 failed, 1 passed of 6	1.52 s	1.6 s
U26	N/A	N/A	N/A	N/A
U27	N/A	N/A	N/A	N/A
U28	444 passed, 1 failed of 445	371 passed, 74 failed of 445	1 min 26.7 s	1 min 28.1 s
U29	44 passed, 1 failed of 45	44 passed, 1 failed of 45	18.6 s	16.7 s
U30	80 passed of 80 total	80 passed of 80 total	29.8 s	11.8 s

Note: The test failures in U28 after repair are due to an obsolete browser. Refer to Section V-C for more details.

TABLE VIII
TEST EXECUTION RESULTS APPLYING LEAKPAIR TO THE SUBJECTS IN TABLE III

ID	Test results before applying LEAKPAIR	Test results after applying LEAKPAIR	Elapsed time before applying LEAKPAIR	Elapsed time after applying LEAKPAIR
K1	N/A	N/A	N/A	N/A
K2	N/A	N/A	N/A	N/A
K3	14 passed of 14	14 passed of 14	41.2 s	35 s
K4	N/A	N/A	N/A	N/A
K5	1610 passed of 1610	1610 passed of 1610	4 s	4 s
K6	64 passed of 64	64 passed of 64	10.01 s	10.2 s
K7	1656 passed of 1750	1656 passed of 1750	1.5 s	1.6 s
K8	275 passed of 277	275 passed of 277	10.582 s	8.549 s
K9	N/A	N/A	N/A	N/A
K10	101 passed of 101	101 passed of 101	3.685 s	3.885 s
K11	62 passed of 62	62 passed of 62	3.7 s	2.2 s
K12	N/A	N/A	N/A	N/A
K13	7 passed of 7	7 passed of 7	2.2 s	2 s
K14	N/A	N/A	N/A	N/A
K15	64 passed of 64	64 passed of 64	10.6 s	8.6 s
K16	373 passed, 11 failed of 384	373 passed, 11 failed of 384	19.4 s	21.1 s
K17	124 passed of 124	124 passed of 124	10.7 s	11.7 s
K18	244 passed of 244	244 passed of 244	30.1 s	30.9 s
K19	N/A	N/A	N/A	N/A
K20	21 passed of 21 total	21 passed of 21 total	5 s	5.7 s
K21	181 passed, 67 skipped of 248	179 passed, 69 skipped of 248	21.4 min	22.6 min
K22	163 passing, 3 failing of 166	163 passing, 3 failing of 166	3 min	3 min
K23	132 passed of 132	132 passed of 132	0.2 s	0.2 s
K24	9 passed, 1 failed of 10	9 passed, 1 failed of 10	37.9 s	67.2 s
K25	192 passed of 192	192 passed of 192	10 s	9.2 s
K26	N/A	N/A	N/A	N/A
K27	178 passed of 178	178 passed of 178	3.4 s	1.7 s
K28	1 passed of 1	1 passed of 1	2.8 s	4.5 s
K29	N/A	N/A	N/A	N/A
K30	N/A	N/A	N/A	N/A

TABLE IX
MEMORY CONSUMPTION RESULTS BEFORE AND AFTER APPLYING GPT-4 TO THE SUBJECTS IN TABLE III. IN THE FIFTH COLUMN, WE SHOW THE HEAP SIZES FOR N PATCHED VERSIONS, WHERE N IS EQUAL TO THE NUMBER OF PLAUSIBLE PATCHES (THE 7TH COLUMN) MINUS THE NUMBER OF PLAUSIBLE PATCHES THAT CAUSE MEMLAB FAILURES (THE 10TH COLUMN). WE USE THE NOTATION X / Y, WHERE X AND Y REPRESENT THE MEAN VALUE AND STANDARD DEVIATION, RESPECTIVELY. IN THE LAST COLUMN, WE HIGHLIGHT THE CASES WHERE LEAKPAIR OUTPERFORMS GPT-4 IN TERMS OF HEAP REDUCTION

Project ID	Heap Size (Original Version)	File ID	Patch Granularity	Heap Size (Patched Versions)	Heap Size Reduction	Plausible Patches	Valid Patches	Run Failure	MemLab Failure	Test Fails	Heap Size Reduction (LEAKPAIR)
K1	28.79 MB	1	Function	28.86 MB / 0.23	-0.07 MB (-0.2%)	10	1	0	0	0	0 MB (0%)
K2	58.53 MB	1	File	58.8 MB / 0.33	-0.27 MB (-0.5%)	6	1	4	0	0	0.2 MB (0.1%)
K3	The context window size is exceeded.										0.1 MB (1.4%)
K4	The context window size is exceeded.										0 MB (0%)
K5	11.2 MB	1	File	-	-	0	0	10	0	0	0.1 MB (0.1%)
K6	13.91 MB	1	File	14.03 MB / 0.18	-0.12 MB (-0.9%)	8	1	2	0	0	0 MB (0%)
K7	9.82 MB	1	Function	9.81 MB / 0.03	0.01 MB (0.1%)	10	9	0	0	0	0.1 MB (0.2%)
K8	64.95 MB	1	File	65.79 MB / 1.5	-0.84 MB (-1.3%)	8	2	2	0	0	0.3 MB (0.2%)
		2	File	63.45 MB / 2.82	1.5 MB (2.3%)	10	6	0	0	0	
K9	31.11 MB	1	Function	31.4 MB / 0.45	-0.29 MB (-0.9%)	8	3	2	0	0	0 MB (0%)
K10	10 MB	1	File	10 MB / 0	0 MB (0%)	3	0	7	0	0	0.0 MB (0%)
		2	File	10 MB / 0	0 MB (0%)	10	0	0	0	0	
		3	File	10 MB / 0	0 MB (0%)	10	0	0	0	0	
		4	File	10 MB / 0	0 MB (0%)	10	0	0	0	0	
K11	66.2 MB	1	File	-	-	10	0	0	10	0	1 MB (0%)
		2	Function	66.96 MB / 0.24	-0.76 MB (-1.1%)	10	0	0	1	0	
K12	6.66 MB	1	Function	6.68 MB / 0.04	-0.02 MB (-0.3%)	10	2	0	1	0	12.3 MB (21.6%)
K13	98.21 MB	1	Function	98.21 MB / 0.03	0 MB (0%)	10	9	0	0	0	0 MB (0%)
K14	26.77 MB	1	Class	26.75 MB / 0.07	0.02 MB (0.1%)	2	1	8	0	0	0.0 MB (0%)
K15	33.71 MB	1	File	33.76 MB / 0.05	-0.05 MB (-0.1%)	9	3	1	1	0	0.8 MB (2.4 %)
		2	File	34.2 MB / 0.28	-0.49 MB (-1.5%)	4	0	6	2	0	
		3	File	-	-	0	0	10	0	0	
		4	File	-	-	0	0	10	0	0	
		5	File	33.71 MB / 0.03	0 MB (0%)	9	8	1	0	0	
K16	33.53 MB	1	Function	33.34 MB / 0.22	0.19 MB (0.6%)	9	5	0	4	1	0.1 MB (0.2%)
K17	The context window size is exceeded.										0 MB (0%)
K18	The context window size is exceeded.										0 MB (0%)
K19	20.48 MB	1	Class	20.32 MB / 0.06	0.16 MB (0.8%)	10	9	0	0	0	0.2 MB (3.3%)
K20	There is no atomic patch within the file granularity.										0 MB (0%)
K21	263.65 MB	1	Class	The context window size is exceeded.							0.1 MB (0.03%)
		2	Class	-	-	0	0	0	5	10	
K22	55.4 MB	1	Class	55.4 MB / 0	0 MB (0%)	10	0	0	0	0	0.5 MB (0.9%)
K23	51.8 MB	1	Class	52.18 MB / 0.04	-0.38 MB (-0.7%)	10	0	0	1	0	0.3 MB (0.3%)
		2	Class	52.2 MB / 0	-0.4 MB (-0.8%)	10	0	0	0	0	
K24	9.2 MB	1	Function	9.6 MB / 0	-0.4 MB (-4.3%)	10	0	0	5	0	0 MB (0%)
K25	The context window size is exceeded.										0 MB (0%)
K26	20.42 MB	1	Class	20.4 MB / 0	0.02 MB (0.1%)	3	3	0	1	7	0 MB (0%)
K27	The context window size is exceeded.										1.6 MB (1.1%)
K28	33.7 MB	1	Class	33.7 MB / 0	0 MB (0%)	10	0	0	0	0	0 MB (0%)
K29	8.3 MB	1	Function	8.36 MB / 0.13	-0.06 MB (-0.7%)	10	0	0	0	0	0 MB (0%)
		2	Class	8.39 MB / 0.17	-0.09 MB (-1.1%)	10	0	0	0	0	
K30	21.48 MB	1	File	20.8 MB / 0.57	0.68 MB (3.2%)	10	3	0	6	0	2.8 MB (45.3%)
Total						259	66	63	37	18	

page; for example, an error occurs when the scenario file refers to a UI element that does not exist on the web page.

We measure the heap size for the plausible patches. The fifth column shows the mean value (shown before '/') and standard deviation (shown after '/') of the heap size observed when the plausible patches are applied. If MemLab fails to run on a patched project, we exclude that problematic patch from the heap size measurement. In the "Heap Size Reduction" column, we compare the second and fifth columns; both the reduced heap size and reduction ratio are shown. Note that a negative value represents an increase in the heap size after applying the patch. Finally, the last column shows the results of LEAKPAIR, which are the same as those in Table V. This column is to compare the results of LEAKPAIR with those of GPT-4.

The performance of GPT-4 is not as good as that of LEAKPAIR. When comparing the sixth and last columns, LEAKPAIR reduces the heap size more than GPT-4 in 23 subjects out of 30 subjects. In the table, we highlight the subjects where LEAKPAIR outperforms GPT-4 in terms of heap size reduction. Recall that we consider a patch valid if it passes the test suite and reduces the heap size. Out of 340 generated patches, approximately 19% of them (66) are found valid. Out of 30 subjects, 14 have at least one valid patch.

We inspect the qualitative aspect of the patches generated by GPT. For example, the patch generated from GPT-4, shown in Fig. 8(a), is almost identical with the developer-written patch shown in Fig. 8(c). However, there is a subtle difference between the two patches: the developer-written patch includes the `capture` option in the first two `removeEventListener` calls, whereas the GPT-4-generated patch does not. The patch generated by GPT-4 does not remove the event listeners correctly, potentially leading to memory leaks. When `removeEventListener` is called, it removes the listener whose event type (the first argument), listener function (the second argument) and

```
+ this.$refs.objectViewWrapper.removeEventListener('dragover', this.onDragOver);
+ this.$refs.objectViewWrapper.removeEventListener('drop', this.editIfEditable);
+ this.$refs.objectViewWrapper.removeEventListener('drop', this.addObjectToParent);
```

(a) GPT-generated patch

```
+ this.$refs.objectViewWrapper.removeEventListener('dragover', this.onDragOver, { capture: true });
+ this.$refs.objectViewWrapper.removeEventListener('drop', this.editIfEditable, { capture: true });
+ this.$refs.objectViewWrapper.removeEventListener('drop', this.addObjectToParent);
```

(b) LEAKPAIR-generated patch

```
  this.$refs.objectViewWrapper.addEventListener('dragover', this.onDragOver, { capture: true });
  this.$refs.objectViewWrapper.addEventListener('drop', this.editIfEditable, { capture: true });
  this.$refs.objectViewWrapper.addEventListener('drop', this.addObjectToParent);
  ...
+ this.$refs.objectViewWrapper.removeEventListener('dragover', this.onDragOver, { capture: true });
+ this.$refs.objectViewWrapper.removeEventListener('drop', this.editIfEditable, { capture: true });
+ this.$refs.objectViewWrapper.removeEventListener('drop', this.addObjectToParent);
```

(c) Developer-written patch

Fig. 8. Comparison of patches produced by GPT-4, LeakPair, and the developer for the second file of K21.

options (the third argument) all match. The first three lines of Fig. 8(c) show how the event listeners are added using the `addEventListener` method. Note that the `capture` option is set to true in the first two `addEventListener` calls. Since the first two `removeEventListener` calls in Fig. 8(a) do not match the added listeners, they fail to remove them. In our experiments, GPT-4 produced the correct patch in only 3 out of 10 trials, despite the fact that the three `addEventListener` calls are included in the prompt. In contrast, LEAKPAIR successfully generates the correct patch, as shown in Fig. 8(b), using the FP2 fix pattern. Recall that the FP2 fix pattern ensures that the arguments passed to `removeEventListener` match those used in the corresponding `addEventListener` calls.

As another example, consider Fig. 9, which shows the patch for the K24 subject. Memory leaks occur because the `Interval` handles created in Line 5 of Fig. 9(a) are not cleared. The developer-written patch shown in Fig. 9(a) calls `clearInterval` in Line 3 to clear the interval handle when the enclosing component is unmounted. LEAKPAIR generates a patch using the fix pattern FP3b. As shown in Fig. 9(b), the obtained patch is semantically the same as the developer-written patch. In contrast, the GPT-4-generated patch shown in Fig. 9(c) adds a call to `clearInterval` in a newly defined function `destroy`. However, this patch, while similar to the developer-written one, does not actually invoke the destroy function, thereby failing to clear the `Interval` handle. As a result, the GPT-4-generated patch adds an unused variable `intervalHandle` and an unused function `destroy`, which results in additional memory consumption without fixing the memory leak. GPT-4 often generates seemingly plausible yet incorrect patches, as shown in the examples above.

Answer to RQ4: *GPT-4 is not as effective as* LEAKPAIR *in generating valid patches for memory leaks in SPAs.* LEAKPAIR *outperforms GPT-4 in terms of heap size reduction for 23 subjects out of the 30 subjects we investigated.*

```
1  setup() {
2     ...
3  + onBeforeUnmount(() =>
       clearInterval(intervalHandle));
4     ...
5  - setInterval(() => { nextSlide() }, slideSpeed)
6  + let intervalHandle = setInterval(() => {
       nextSlide() }, slideSpeed)
7     ...
8  }
```

(a) Developer-written patch.

```
1  setup() {
2     ...
3  + const intervalID = ref();
4  + onBeforeUnmount(() =>
       clearInterval(intervalID.value));
5     ...
6  - setInterval(() => { nextSlide() }, slideSpeed)
7  + intervalID.value = setInterval(() => {
       nextSlide() }, slideSpeed)
8     ...
9  }
```

(b) LEAKPAIR-generated patch.

```
1  setup() {
2     ...
3  - setInterval(() => { nextSlide() }, slideSpeed)
4  + let intervalHandle = setInterval(() => {
       nextSlide() }, slideSpeed)
5     ...
6  + const destroy = () => {
       clearInterval(intervalHandle) }
7     ...
8  }
```

(c) GPT-generated patch.

Fig. 9. Comparison of patches produced by the developer, LeakPair, and GPT-4 for K24.

VI. DISCUSSION

A. Revisiting the Pull Requests

Looking at the response of the live study (PR submissions), it is safe to imply that memory leak issues are deemed critical by the developers; they will readily fix the leaks given that the actual root cause is identified. Even with no significant heap

reduction, developers remain committed to fixing the leak as long as the source is identified. This is because most developers understand that as their application grows, so will the impact of the leak, even if it seems benign at the current stage.

The temporal difference in the creation dates of the PRs (Vue in 2024 vs. Angular and React in 2023) likely affects their approval rates, Vue PRs, being newer, have faced higher unapproval rates (80%) likely due to insufficient review time.

The analysis of the unapproved PRs reveals that 60% of them involve changes in more than five files. This suggests a trend where PRs that modify numerous files could make them more challenging to review, leading to a greater likelihood of unapproval or prolonged review periods.

Another significant observation is that 40% of the unapproved PRs come from projects with fewer than 10 contributors. This statistic implies that larger projects, which typically have more than 10 contributors, may benefit from better-established review processes and more resources for thorough evaluations.

It is important to note that all the unapproved PRs were either ignored or closed due to inactivity. None of these PRs were explicitly rejected based on the quality of the patches themselves.

B. Intrusive Memory Leak Repair (The Case of Uncleared Collections)

During the leak-pattern mining process, we found the use of module/file-scoped collections (sets, arrays) to be quite prevalent in SPAs [110], [111], [112], [113], which happened to be one of the common causes of memory leakage. However, in SPAs, when a variable is defined outside of a function (within the file scope), it is unclear when it will no longer be needed. This causes them to be ignored by the garbage collector, and they remain in memory long after they are no longer needed. This may not be an issue for primitive data types such as strings or integers, but if these variables hold large arrays, their accumulation over time can easily lead to huge memory leaks.

In SPA frameworks, each 'page' or view is rendered by a separate function or class component. When a SPA executes, the views (components) are rendered according to the user's actions. If a component is written in a file that contains module-scoped collection(s), then, even after the component unmounts, the collections defined in the file scope still remain in memory, unless they are also explicitly cleaned up in the destructors of the invoked components.

The two potential approaches for cleaning up these module-level collections are:

1) Clearing them in the destructors of all the components defined in the file.
2) Moving the collection from the module (file) scope to the component scope (in a class constructor or function's local scope); this way they can be dropped by garbage collection when the class instance is dropped or the function control is returned.

The first approach is vulnerable to creating intrusive patches; it may breaks test cases/functionality of the target SPA. The module-scoped variables are accessible to all components in the module, which means there will only ever be one instance of that variable. If there are multiple instances of the component or multiple components in the same file, they will all be sharing the value of that variable, so if we clear the collection variable in the destructors of the component, then the destruction of one component will affect other instances of the component or other components in the same file.

The second approach may lead to redundant memory consumption. While moving collection from module scope to component scope will ensure garbage collection upon class instance destruction or return of function control, there will be a lot of code duplication. As mentioned before, a module (file) may contain more than one component; moving the collection to a component means duplicating it in every component defined in the file. Also, some SPA frameworks, such as React, maintain a component *state*, and the component is re-rendered every time the state changes, which means these huge collections would be reinitialized every time the component state updates, gobbling up a lot of memory and slowing the app's performance.

We believe some trade-off between memory leak prevention and code bloating is inevitable if we consider the second fix, since it still preserves functionality but may impact execution times (performance). Hence, for our current study, we did not incorporate this pattern, as the aim of our approach is to automate simple, non-intrusive fixes that neither impact behaviour nor the performance of the application in any way.

C. Automating Pattern Extraction

I n this work, we manually extracted fix patterns from real patches to fix memory leaks in SPAs. Considering the demonstrated effectiveness of our pattern-based approach, one viable future direction is to automate the extraction of fix patterns. For instance, common fix patterns could be mined from a large number of patches that fix memory leaks in SPAs; similar approaches was used for general program repair tasks [14]. Alternatively, an LLM can be used to fix memory leaks in SPAs and then extract fix patterns from the generated patches. However, the poor performance of GPT-4 in our evaluation questions the feasibility of this approach.

D. Threats to Validity

Threats to external validity may lie in the target subjects that this study uses as they are open-source projects; thus, the results may not be representative of projects, such as those using closed-source techniques. In addition, our study focuses only on JavaScript subjects, while there are other languages implementing SPAs. This threat might be mitigated since our target SPA frameworks (i.e., React and Angular) are popular and representative in the web development community. A nother related threat is that the fix patterns we extracted may work well only for the subjects we used in the evaluation. To mitigate the potential risk of overfitting patterns, we separated the dataset used for pattern extraction (see Section III-B) from the dataset used to evaluate the efficacy of the patterns (see Section IV-B).

Threats to internal validity may include the fix patterns manually extracted by the authors. To address this threat, each

fix pattern is extrapolated from real patches for memory leaks in SPAs.

Threats to construct validity may relate to the test cases used in the evaluation. To show the non-intrusiveness of the patches generated by LEAKPAIR, our experiment runs test cases given by each subject. Although test suites may not prove the correctness of the behavior in the applications, it might be enough to preserve major functionalities in the applications from the maintenance perspective. Since we used the regression test suites provided by the subjects, each subject may have different levels of test coverage. While adding more test cases would be desirable, we found it challenging to add new tests systematically without having domain-specific knowledge on the subjects. To mitigate this threat, we also ran manual tests on the subjects to check for any unexpected behavior but did not find any.

E. Current Limitations

The current implementation of LEAKPAIR applies fix patterns for individual files and does not handle memory leaks whose fix require changes across multiple files. For example, if a fix requires changes in the current file and another file that is imported by the current file, LEAKPAIR cannot generate a patch. As another example, memory allocation and deallocation may occur in different files, and LEAKPAIR cannot generate a patch in this case.

In SPA frameworks, state management and component lifecycles introduce additional complexity. In Angular-based SPAs, for instance, dynamic component creation using `ViewContainerRef` or `NgComponentOutlet` can lead to memory leaks if components are instantiated but not properly destroyed. Since LEAKPAIR currently analyzes files individually, it does not track how components interact dynamically or how services manage shared states across modules. This means that memory leaks caused by cross-module communication, dynamically loaded components, or third-party library integrations are not yet addressed by our implementation.

However, these are implementation limitations rather than fundamental limitations of our approach. It is feasible to extend LEAKPAIR by defining and applying cross-file fix patterns, allowing it to detect and resolve memory leaks that span multiple files.

VII. RELATED WORK

In this section, we will briefly go over the research efforts done in the area of automated memory leak detection in JavaScript as well as the progression of the applications of pattern-based automated program repair over the years, what limitations they encounter and how our approach fares in comparison with these techniques.

A. Memory Leak Detection in JavaScript

There have been a number of studies and proposed approaches presented for the diagnosis and automated detection of memory leaks in JavaScript, however, they all suffer from certain limitations, which leaves the state-of-the-art still being the manual analysis of heap snapshots captured at different points in time. In this section, we will summarize and discuss 5 such leak-detection techniques proposed in the last 7 years, and their corresponding limitations.

Diagnostic information on DOM objects also helps identify the source of leaking objects. Jensen et al. presented MemInsight [9] in 2015, which makes use of modern browser elements to provide comprehensive diagnostic information regarding DOM elements. It leverages the Jalangi framework to instrument the source code to produce traces where the memory of objects causing the leaks is allocated (call trees) and accessed (access paths). The *call tree* shows the context of method calls that assign the leaking object, while *access paths* define the series of objects that contain the leaking object, keeping it from being swept by the garbage collector. MemInsight makes use of a unique object lifetimes analysis, including an advanced DOM modelling mechanism, to gauge the time since the object has gone stale, without leveraging JavaScript's garbage collector.

This tool, however, fails to provide the exact locations of the root cause of the leak in the source code; the developer still has to go through the detailed information and reports provided by MemInsight to identify the actual source of the leak. Moreover, as we explained earlier, approaches based on staleness are unreliable for memory leak detection, as leaking objects could still have unwanted references, preventing them from going *stale*.

Another work [11] was published in the following year by M. Rudafshani and P. A. War, based on the same criteria of leak detection; i.e. object staleness. The tool, called LeakSpot makes use of a run-time heap model by modifying the application code in a browser-agnostic way to record object allocations, accesses, and references. To find the leaked objects and problematic locations in the code, LeakSpot groups objects based on their allocation sites (where in the code objects are allocated) or reference sites (where in the code a reference is created to the objects). LeakSpot refines the allocation sites by making an allocation-site graph. It then determines whether the group of objects are leaked or not based on their corresponding collective-staleness graphs.

To facilitate debugging and fixing the leaks, for every leaked object, LeakSpot reports all the locations in the code where the forgotten references were created. An empirical study conducted by J. Vilk and E. D. Berger [12] revealed that on real web applications, LeakSpot typically reports over 50 different allocation and reference sites which developers then have to manually analyse in order to identify the root cause of the leak.

A variant of leak detection strategy employs the *growth-based* approach. This approach considers the growing usage of heap memory as an indication of leaking memory. J. Qian presented a lightweight approach [10] for memory leak diagnosis in web applications using this criterion. The proposed technique obtains a sequence of heap snapshots by executing the program. These snapshots are parsed and the object reference graphs embedded in these snapshots are traversed and compared to locate objects that are newly created.

The newly created objects are groups based on the Similarity Object Count (SOC) heuristic, where each common parent object in the similarity groups represents a candidate leak root. The memory growth of the candidate leak roots in the heap snapshot is analysed; if the occupied memory consistently grows, then that candidate object is regarded as suspicious, otherwise discarded from the results. The candidate leak roots are ordered by their occupied memory.

There are 2 limitations to this approach; first, it is a leak detection method that is entirely manual and extensive. Second, the candidate causes of leaks are obtained by gauging the growth of objects across heap snapshots, however, growth-based analysis is not always a valid criterion as some growth is expected and desirable such as that in the cache.

While taking a deviation from growth-based and staleness-based approaches, Vilk and E. D. Berger attributed sustained growth of heap between round trips to the same location in the website, as a gauge for leaking memory. They developed BLeak [12], an automated leak detection tool whose algorithm is based on the notion that web app users often return to the same visual state after performing some actions. The rationale is that visiting the same visual state should consume almost the same amount of memory, therefore, if there is sustained growth in memory consumption (growing objects) between the loops to the same state, it is a valid indicator of memory leakage.

BLeak first uses heap differencing to locate locations in a heap with sustained growth between each round trip, which it identifies as leak roots. To directly identify the root causes of growth, BLeak employs JavaScript rewriting to target leak roots and collect stack traces when they grow. Finally, when presenting the results to the developer, BLeak ranks leak roots by return on investment using a novel metric called LeakShare that prioritizes memory leaks that free the most memory with the least effort.

However, since it relies on interaction with the website, BLeak requires a scenario file written on the part of the user, to be able to run the web app in a headless browser, specifying the steps to complete the round trip. In addition, it takes around 10 minutes to execute. These 2 factors in our opinion are a major hindrance in the prevalent usage of this tool.

The latest dynamic approach to leak detection (at the time of writing), was introduced in late 2022, as Memlab [44], by team Meta at Facebook. Memlab reports retainer traces of memory leaks by running the web app in a headless browser. For that, it needs a scenario file written by the user, just as in the case of BLeak. Similar to BLeak, the scenario file must contain steps that complete a full round trip of web interaction.

For each group of leaked objects, Memlab prints one leak trace, called the retainer trace. The trace is an object reference from the GC root to the leaked objects. However, the trace, just like the heap snapshot, is interposed with metadata such as V8 HiddenClasses and class prototypes. The idea is that if the user follows the trace from the root to the final leaked object, they should be able to identify the unwanted reference that should be released (set to null) to break the chain to the root, thereby fixing the leak.

All the aforementioned approaches are to detect memory leaks in JavaScript applications. While these tools often provide debugging information to help developers identify the root cause of the leak, they do not provide automated fixes. In contrast, our approach proactively repairs memory leaks in SPAs without requiring explicit leak detection.

B. Pattern Based Program Repair

The idea of automated program repair based on recurring patterns mined from real-world projects was first proposed by Kim et al. [16] in 2013. The authors manually reviewed 60,000 patches from real-world projects, curated the recurring fix patterns and applied them as what was then termed Patch-Based Automated Program Repair. The approach was evaluated on 253 subjects by comparing the patterns generated by PAR with those of GenProg [36]. Patches generated by PAR were shown to have a higher ratio of acceptance by subjects' developers.

However, in contrast to LEAKPAIR's straightforward approach of direct application of non-intrusive patches, PAR depends on first localizing the statements to modify by statistical fault localization and modifies only those statements that are visited by failing test cases.

There also have been efforts to employ pattern-based repairs to repair DOM-related bugs. Vejovis [41], introduced in a study published in 2014, is one such tool. The study analyzed 190 real-world bug reports to detect recurring fixes to DOM-related faults. The 2 most common fix categories were found to be parameter replacements and DOM element validations, which were then automated in the said tool. Vejovis was evaluated on 11 subjects, on a total of 22 real-world bugs. Vejovis was successfully able to repair 20 out of those 22 bugs, 65% of which were ranked top by the tool as the correct fix.

While this approach is limited to 2 specific DOM-related bugs, our approach is able to address the general issue of memory leaks, and can be extended to any memory leak pattern so long the fix pattern is non-intrusive.

Pattern-based repairs have been applied to target performance-related bugs such as high resource consumption. Caramel [35], introduced in 2015, leverages non-intrusive fix patterns, that are simple, easy to understand, and easily acceptable to developers to address redundant computations of loops, which wastes computations and memory. The fix was fairly straightforward: breaking out of the loop as soon as the condition became true. Caramel provides a source-level patch to the user for each bug. The tool was evaluated on 11 and 4 popular Java C/C++ applications respectively. The tool was able to identify 150 unknown performance bugs across all subjects, and successfully generate fixes for 149 out of those. 77.3% of the bugs were fixed by developers, at the time of the publication. However, unlike LEAKPAIR, the fixes generated by Caramel are documented in a bug report rather than directly applied in the source code.

Miscellaneous bugs have also been addressed by leveraging common patterns. In 2018, Liu et al. [42] presented an approach that extracted code samples from StackOverflow and then mined 13 fix patterns from them. The fix templates were

implemented in a tool called SOfix, which was evaluated on the Defects4J benchmark. The tool was able to repair 23 bugs, which, at the time of the study, was the highest count of automatically repaired bugs among the contemporary approaches.

The patterns in SOfix, however, are only derived from StackOverflow posts. In contrast, our approach ensures the validity of the patterns by mining them from merged commits on GitHub. In addition, the bug patterns targeted by SOfix are miscellaneous, while LEAKPAIR focuses on improving the performance of the subject by mitigating overall memory leaks in the application.

In the same year, Liu et al proved the effectiveness and efficacy of pattern-based program repair in a study [15], by developing TBar, a simple template-based APR tool that applies recurrently used fix patterns on already localized, miscellaneous bugs. The evaluation was done on the Defects4J benchmark, and the tool was successfully able to repair 74 out of 101 localized bugs. This, at the time of the study, was a record performance by a Java APR tool, and the authors expect the tool to be regarded as a baseline for further developments in the domain of pattern-based program repair.

TBar, however, does not aim to improve a particular functional or non-functional aspect of the application. Our approach, in contrast, targets the performance aspect of the application i.e. memory usage, and was shown to be successful in improving the state of memory leakage of the applications.

Efforts have also been made to automate the process of mining patterns itself. In 2019, Koyuncu at al. proposed FixMiner [14] a tool for automatically mining fix patterns, leveraging a 'three-fold iterative clustering' strategy, which can then be utilized by automated patch generation tools. Using the AST context of the code changes, it tries to cluster the recurring changes based on their similarity. The tool and its mined patches were evaluated on already-curated 1000 bugs. FixMiner was shown to curate accurate and effective patches from open-source projects. Furthermore, an APR prototype, PARFixMiner, was developed to implement the patterns curated by FixMiner. PARFixMiner was successfully able to fix 26 Defects4J benchmark bugs, Moreover, 81% of FixMiner's generated patches were proved to be correct, showing a high probability of their correctness.

Though shown to curate accurate and effective patches, it does not address the automated repair of bugs. LEAKPAIR on the other hand, does not have automated mining capability but provides automated repair without requiring prior bug localization. We believe these two tools to be complementary to each other, if not directly comparable.

C. Pattern Based APR Requiring Fault Localization

Some pattern based repair tools rely on the localization of the bugs to be able to generate the corresponding patch. Jiang et al's SimFix [43] is one such tool, that was introduced in 2018. SimFix leverages both existing patches and similar code to automatically repair programs. The intersection of these 2 search spaces was then searched to find the final fix patterns for the localized faults. SimFix was evaluated on the Defects4J benchmark and was able to fix 34 bugs which, at the time of the study was the largest count of bugs fixed by an APR tool on the benchmark, surpassing SOFix. Furthermore, 13 of the fixes were fixed for the first time by any APR tool.

Another APR tool assuming fault localization was proposed in the following year by K. Liu et al, named as AVATAR [50]. The tool leverages patches of static code analysis violations to generate its own fixes since such patches have already been systematically assessed by the static detectors and can be relied on. AVATAR assumes an accurate localization of fault, to be able to generate the the corresponding patch. The tool was evaluated on the Defects4J benchmark and was successfully able to repair 34 of 39 localized bugs. The performance was compared with the contemporary approaches and was found to outperform most of them while still being complementary to most.

Both of these approaches depend on an ordered list of suspicious faulty statements using standard fault localization approaches. Our approach, on the other hand, provides a big edge over such approaches as it is able to repair bugs without the prerequisite of localized faults.

D. Usability of Automated Patches

Tao et al. [114] evaluate the impact of auto-generated patches on debugging effectiveness. In a large-scale human study involving 95 participants, the study found that high-quality patches significantly improved debugging correctness. Participants appreciated the quick problem identification provided by generated patches but doubted their effectiveness for complex bugs. The findings emphasize the impracticality of the direct deployment of automated patches due to readability and maintainability concerns. Instead, the authors argue that such auto-generated patches can be useful in aiding debugging by suggesting potential fixes. Our live study (patch submissions) results, on the other hand, reveal that developers are much likely to accept the patches if the fixes are simple, non-intrusive, and have a noticeable impact (in this case, performance improvement).

R2Fix [115] is an automated tool designed to generate patches for software written in C/C++ by analyzing free-form bug reports. Its main goal is to reduce the time and effort required for developers to fix bugs, particularly focusing on buffer overflows, null pointer bugs, and memory leaks. The tool generates patch suggestions automatically, producing an average of 1.33 patches per bug report, and developers can verify and apply these patches directly. Overall, the tool proved its efficacy in shortening the bug-fixing times by up to 63 days. Our tool, on the other hand, focuses on improving the performance (memory utilization) of web applications by proactively repairing memory leaking patterns directly in the source code.

VIII. CONCLUSION

In this work, we have introduced a novel technique LEAKPAIR to fix memory leaks in single page web applications. Despite the prevalence of single-page web applications and their memory leaks, there has been no research effort to fix

those bugs automatically. We have shown that by using only a handful of fix patterns mined from the existing patches, diverse SPAs of 37 open-source projects can be successfully fixed. Furthermore, the patches generated by LEAKPAIR are high-quality (the majority of the pull requests LEAKPAIR made were accepted by the original developers) and safe to accept (the fix patterns we use are non-intrusive).

This work also aims at fixing a specific type of bug, i.e., memory leaks in single-page applications. The proposed technique is simple as compared to recent approaches. However, simplicity does not necessarily imply ineffectiveness. On the contrary, LEAKPAIR is very effective, as was shown. We view this as the strength of our approach. For certain types of bugs, simple pattern-based approaches, like ours, do a good job without using heavy-weight deep learning or implementing complex static analysis and proving the correctness of the analysis.

DATA AVAILABILITY STATEMENT

We make the replication package publicly available, which includes all the code and datasets to reproduce our experiments at https://github.com/Arooba-git/leakpair-study-replication/ [51].

REFERENCES

[1] "roosterjs," Accessed: Feb. 15, 2023. [Online]. Available: https://github.com/microsoft/roosterjs/commit/c3f2f0c4d229502c634e6c99b604df3e5f47b9b6

[2] N. Lazarov, "Memory leaks and memory consumption in web applications (part 1)." Accessed: Jun. 04, 2024. [Online]. Available: https://www.telerik.com/blogs/memory-leaks-and-memory-consumption-in-web-applications-part-1

[3] G. Fink, I. Flatow, and S. E. L. A. Group, *Pro Single Page Application Development: Using Backbone.js and ASP.NET.* New York, NY, USA: Apress, May 2014.

[4] K. Lawson, "What are single page applications and why do people like them so much?," Accessed: Jun. 04, 2024. [Online]. Available: https://www.bloomreach.com/en/blog/2018/what-is-a-single-page-application?spz=article_var

[5] "4 Types of Memory Leaks in JavaScript and How to Get Rid Of Them," Accessed: Feb. 15, 2023. [Online]. Available: https://auth0.com/blog/four-types-of-leaks-in-your-javascript-code-and-how-to-get-rid-of-them

[6] "roosterjs," Accessed: Feb. 15, 2023. [Online]. Available: https://github.com/microsoft/roosterjs

[7] "Window: hashchange event - web APIS | MDN," Accessed: Feb. 15, 2023. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/Window/hashchange_event

[8] V. Azhari, S. Bhamra, N. Ezzati-Jivan, and F. Tetreault, "Efficient heap monitoring tool for memory leak detection and root-cause analysis," in *Proc. IEEE Int. Conf. Big Data (Big Data)*, Orlando, FL, USA. Piscataway, NJ, USA: IEEE Press, 2021, pp. 3020–3030.

[9] S. H. Jensen, M. Sridharan, K. Sen, and S. Chandra, "Meminsight: Platform-independent memory debugging for JavaScript," in *Proc. 10th Joint Meeting Found. Softw. Eng. (ESEC/FSE)*, New York, NY, USA: ACM, 2015, pp. 345–356, doi: 10.1145/2786805.2786860.

[10] J. Qian, L. Wang, and X. Zhou, "A lightweight approach to detect memory leaks in JavaScript (s)," in *Proc. Int. Conf. Softw. Eng. Knowl. Eng.*, San Francisco, California, USA: KSI Res. Inc., 07, 2018, pp. 582–640.

[11] M. Rudafshani and P. A. S. Ward, "Leakspot: Detection and diagnosis of memory leaks in JavaScript applications," *Softw. Pract. Exper.*, vol. 47, no. 1, pp. 97–123, Jan. 2017, doi: 10.1002/spe.2406.

[12] J. Vilk and E. D. Berger, "Bleak: Automatically debugging memory leaks in web applications," *Commun. ACM*, vol. 63, no. 11, pp. 146–153, Oct. 2020, doi: 10.1145/3422598.

[13] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra, "SemFix: Program repair via semantic analysis," in *Proc. Int. Conf. Softw. Eng. (ICSE)*, Piscataway, NJ, USA: IEEE Press, 2013, pp. 772–781.

[14] A. Koyuncu et al., "Fixminer: Mining relevant fix patterns for automated program repair," *Empirical Softw. Engg*, vol. 25, no. 3, pp. 1980–2024, May 2020, doi: 10.1007/s10664-019-09780-z.

[15] K. Liu, A. Koyuncu, D. Kim, and T. F. Bissyandé, "Tbar: Revisiting template-based automated program repair," in *Proc. 28th ACM SIGSOFT Int. Symp. Softw. Testing Anal. (ISSTA)*, New York, NY, USA: ACM, 2019, pp. 31–42, doi: 10.1145/3293882.3330577.

[16] D. Kim, J. Nam, J. Song, and S. Kim, "Automatic patch generation learned from human-written patches," in *Proc. 35th Int. Conf. Softw. Eng. (ICSE)*, San Francisco, CA, USA. Piscataway, NJ, USA: IEEE Press, 2013, pp. 802–811.

[17] C. L. Goues, M. Pradel, and A. Roychoudhury, "Automated program repair," *Commun. ACM*, vol. 62, no. 12, pp. 56–65, 2019.

[18] A. Shahoor, A. Y. Khamit, J. Yi, and D. Kim, "Leakpair: Proactive repairing of memory leaks in single page web applications," in *Proc. 38th IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE)*, Sep. 2023, pp. 1175–1187. [Online]. Available: https://ieeexplore.ieee.org/document/10298488

[19] "Angular," Accessed: Mar. 13, 2024. [Online]. Available: https://angular.io

[20] "React," Accessed: Mar. 13, 2024. [Online]. Available: https://react.dev

[21] "GPT-4," Accessed: Sep. 04, 2024. [Online]. Available: https://openai.com/index/gpt-4/

[22] C. S. Xia and L. Zhang, "Automated program repair via conversation: Fixing 162 out of 337 Bugs for $0.42 Each using ChatGPT," in *Proc. 33rd ACM SIGSOFT Int. Symp. Softw. Testing Anal.* Vienna Austria. New York, NY, USA: ACM, Sep. 2024, pp. 819–831, doi: 10.1145/3650212.3680323.

[23] "Feb, Garbage collection in redux applications," Accessed: Apr. 8, 2024. [Online]. Available: https://developers.soundcloud.com/blog/garbage-collection-in-redux-applications

[24] "Redux - A JS library for predictable and maintainable global state management| Redux," Accessed: Apr. 8, 2024. [Online]. Available: https://redux.js.org

[25] "Introducing fuite: a tool for finding memory leaks in web apps," Accessed: Feb. 15, 2023. [Online]. Available: https://nolanlawson.com/2021/12/17/introducing-fuite-a-tool-for-finding-memory-leaks-in-web-apps

[26] "A tour of V8: Garbage collection," Accessed: Feb. 15, 2023. [Online]. Available: https://jayconrod.com/posts/55/a-tour-of-v8--garbage-collection

[27] "Pomodore-discord-bot," Accessed: Feb. 15, 2023. [Online]. Available: https://github.com/MarcoPereira27/pomodore-discord-bot/issues/4

[28] "Strange Nodejs memory leak," Accessed: Feb. 15, 2023. [Online]. Available: https://stackoverflow.com/questions/63661738/strange-nodejs-memory-leak

[29] "angular," Accessed: Feb. 15, 2023. [Online]. Available: https://github.com/angular/angular/issues/27803

[30] "Solving memory leaks in large react application," Accessed: Feb. 15, 2023. [Online]. Available: https://stackoverflow.com/questions/63813604/solving-memory-leaks-in-large-react-application

[31] "angular," Accessed: Feb. 15, 2023. [Online]. Available: https://github.com/angular/angular/issues/20007

[32] "BloatBusters - WebPerfDays," Accessed: Aug. 31, 2023. [Online]. Available: https://docs.google.com/presentation/d/1wUVmf78gGra5aOxvTfYdiLkdGaR9OhXRnOlIcEmu2s/edit#slide=id.g1d65bdf6_0_0

[33] Taub, C. "How we resolved a memory leak on our website," DEV Community, May 2021. Accessed: Aug. 13, 2023. [Online]. Available: https://dev.to/fiit/how-we-resolved-a-memory-leak-on-our-website-1kf0

[34] Msedgeteam, "The heap snapshot file format - Microsoft Edge Development," Accessed: Aug. 31, 2023. [Online]. Available: https://learn.microsoft.com/en-us/microsoft-edge/devtools-guide-chromium/memory-problems/heap-snapshot-schema

[35] A. Nistor, P.-C. Chang, C. Radoi, and S. Lu, "Caramel: Detecting and fixing performance problems that have non-intrusive fixes," in *Proc. 37th Int. Conf. Softw. Eng. (ICSE)*, vol. 1. Piscataway, NJ, USA: IEEE Press, 2015, pp. 902–912.

[36] W. Weimer, T. Nguyen, C. L. Goues, and S. Forrest, "Automatically finding patches using genetic programming," in *Proc. 31st Int. Conf. Softw. Eng. (ICSE)*, Los Alamitos, CA, USA: IEEE Comput. Soc. Press, 2009, pp. 364–374.

[37] Q. Zhu et al., "A syntax-guided edit decoder for neural program repair," in *Proc. ACM Joint Meeting Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.*, New York, NY, USA: ACM, Aug. 2021, pp. 341–353.

[38] T. Lutellier, H. V. Pham, L. Pang, Y. Li, M. Wei, and L. Tan, "CoCoNuT: combining context-aware neural translation models using ensemble for program repair," in *Proc. 29th ACM SIGSOFT Int. Symp. Softw. Testing Anal.,* Virtual Event USA, New York, NY, USA: ACM, Jul. 2020, pp. 101–114, doi: 10.1145/3395363.3397369.

[39] R. van Tonder and C. L. Goues, "Static automated program repair for heap properties," in *Proc. 40th Int. Conf. Softw. Eng.,* 2018, pp. 151–162.

[40] S. Hong, J. Lee, J. Lee, and H. Oh, "SAVER: Scalable, precise, and safe memory-error repair," in *Proc. ACM/IEEE 42nd Int. Conf. Softw. Eng.,* 2020, pp. 271–283.

[41] F. S. Ocariza, Jr., K. Pattabiraman, and A. Mesbah, "Vejovis: Suggesting fixes for JavaScript faults," in *Proc. 36th Int. Conf. Softw. Eng. (ICSE)*, New York, NY, USA: ACM, 2014, pp. 837–847, doi: 10.1145/2568225.2568257.

[42] X. Liu and H. Zhong, "Mining stackoverflow for program repair," in *Proc. IEEE 25th Int. Conf. Softw. Anal., Evol. Reeng. (SANER)*, Campobasso, Italy. Piscataway, NJ, USA: IEEE Press, 2018, pp. 118–129.

[43] J. Jiang, Y. Xiong, H. Zhang, Q. Gao, and X. Chen, "Shaping program repair space with existing patches and similar code," in *Proc. 27th ACM SIGSOFT Int. Symp. Softw. Test. Anal. (ISSTA)*, New York, NY, USA: ACM, 2018, pp. 298–309, doi: 10.1145/3213846.3213871.

[44] G. C. Liang Gong, "MemLab: An open source framework for finding JavaScript memory leaks," *Eng. Meta*, Oct. 2022. Accessed: Jan. 03, 2025. [Online]. Available: https://engineering.fb.com/2022/09/12/open-source/memlab

[45] "RxJS - Observable," Accessed: Feb. 16, 2023. [Online]. Available: https://rxjs.dev/guide/observable

[46] "RxJS - takeUntil," Accessed: Feb. 16, 2023. [Online]. Available: https://rxjs.dev/api/operators/takeUntil

[47] "Codecademy," Accessed: Feb. 17, 2023. [Online]. Available: https://www.codecademy.com/courses/react-101/lessons/component-lifecycle-methods/exercises/componentwillunmount

[48] Window.requestAnimationFrame() - Web APIs | MDN. Accessed: Feb. 17, 2023. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/window/requestAnimationFrame

[49] "Events API | Vue 3 Migration Guide," Accessed: Jan. 4, 2024. [Online]. Available: https://v3-migration.vuejs.org/breaking-changes/events-api.html

[50] K. Liu, A. Koyuncu, D. Kim, and T. F. Bissyande, "Avatar: Fixing semantic bugs with fix patterns of static analysis violations," in *Proc. IEEE 26th Int. Conf. Softw. Anal., Evol. Reeng. (SANER)*, Los Alamitos, CA, USA: IEEE Computer Soc., Feb. 2019, pp. 1–12. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/SANER.2019.8667970

[51] "Mar, leakpair-study-replication," 2024. Accessed: Mar. 13, 2024. [Online]. Available: https://github.com/Arooba-git/leakpair-study-replication/tree/main

[52] "Babel · The Compiler for Next Gener. JavaScript," vol. 17, p. 2023, Accessed: Jul. 13, 2022. [Online]. Available: https://babeljs.io

[53] "jscodeshift," Accessed: Feb. 16, 2023. [Online]. Available: https://github.com/facebook/jscodeshift

[54] "recast," Accessed: Feb. 16, 2023. [Online]. Available: https://github.com/benjamn/recast

[55] "react-zoom-pan-pinch," Accessed: Feb. 15, 2023. [Online]. Available: https://github.com/prc5/react-zoom-pan-pinch/pull/270/commits

[56] "angular-components," Accessed: Feb. 16, 2023. [Online]. Available: https://github.com/angular/components

[57] "evergreen," Accessed: Feb. 16, 2023. [Online]. Available: https://github.com/segmentio/evergreen

[58] "ngx-datatable," Accessed: Feb. 16, 2023. [Online]. Available: https://github.com/swimlane/ngx-datatable

[59] "react-multi-carousel," Accessed: Feb. 16, 2023. [Online]. Available: https://github.com/YIZHUANG/react-multi-carousel

[60] "angular-UI," Accessed: Feb. 16, 2023. [Online]. Available: https://github.com/DetektivKollektiv/angular-ui

[61] "retail-UI," Accessed: Feb. 16, 2023. [Online]. Available: https://github.com/skbkontur/retail-ui/tree/retail-ui%401.11.1

[62] "NDB-core," Accessed: Feb. 16, 2023. [Online]. Available: https://github.com/Aam-Digital/ndb-core

[63] "DevTools," Accessed: Feb. 16, 2023. [Online]. Available: https://github.com/replayio/devtools

[64] "ngx-bootstrap," Accessed: Feb. 16, 2023. [Online]. Available: https://github.com/valor-software/ngx-bootstrap

[65] "DefiChain-income," Accessed: Oct. 1, 2023. [Online]. Available: https://github.com/rogi-sh/defichain-income

[66] "collosal," Accessed: Oct. 1, 2023. [Online]. Available: https://github.com/iceboy1406/collosal

[67] "tbt-website," Accessed: Oct. 1, 2023. [Online]. Available: https://github.com/tbtMEC/tbt-website

[68] "mempool," Accessed: Sep. 30, 2023. Available: https://github.com/mempool/mempool

[69] "DSpace-angular," Accessed: Oct. 1, 2023. [Online]. Available: https://github.com/DSpace/dspace-angular

[70] "PrimeNG," Accessed: Oct. 1, 2023. [Online]. Available: https://github.com/primefaces/primeng

[71] "NGX-formly," Accessed: Oct. 1, 2023. [Online]. Available: https://github.com/ngx-formly/ngx-formly

[72] "studio," Accessed: Oct. 1, 2023. [Online]. Available: https://github.com/foxglove/studio

[73] "BootstrapVue," Accessed: Jan. 1, 2024. [Online]. Available: https://github.com/bootstrap-vue/bootstrap-vue

[74] "chatwoot," Accessed: Jan. 1, 2024. [Online]. Available: https://github.com/chatwoot/chatwoot

[75] "think-vuele," Accessed: Jan. 1, 2024. [Online]. Available: https://github.com/chfree/think-vuele

[76] "vue-admin-better," Accessed: Jan. 1, 2024. [Online]. Available: https://github.com/chuzhixin/vue-admin-better?tab=readme-ov-file

[77] "vue-grid-layout," Accessed: Jan. 1, 2024. [Online]. Available: https://github.com/jbaysolutions/vue-grid-layout

[78] "weaverbird," Accessed: Jan. 1, 2024. [Online]. Available: https://github.com/ToucanToco/weaverbird

[79] "auto-animate," Accessed: Jan. 1, 2024. [Online]. Available: https://github.com/formkit/auto-animate

[80] "vue-snap," Accessed: Jan. 1, 2024. [Online]. Available: https://github.com/bartdominiak/vue-snap

[81] "element," Accessed: Jan. 1, 2024. [Online]. Available: https://github.com/ElemeFE/element

[82] "lan-ui," Accessed: Jan. 1, 2024. [Online]. Available: https://github.com/lan-ui/lan-ui

[83] "iVIEW," Accessed: Jan. 1, 2024. [Online]. Available: https://github.com/iview/iview

[84] "buefy," Accessed: Jan. 1, 2024. [Online]. Available: https://github.com/buefy/buefy

[85] "fundamental-NGX," Accessed: Feb. 16, 2023. [Online]. Available: https://github.com/SAP/fundamental-ngx

[86] "material-UI," Accessed: Feb. 16, 2023. [Online]. Available: https://github.com/mui/material-ui

[87] "material.angular.io," Accessed: Feb. 16, 2023. [Online]. Available: https://github.com/angular/material.angular.io

[88] "octant," Accessed: Feb. 16, 2023. [Online]. Available: https://github.com/vmware-archive/octant

[89] "transloco," Accessed: Feb. 16, 2023. [Online]. Available: https://github.com/ngneat/transloco/pull/65/files

[90] "Fix memory leaks by Louptheron · Pull Request #953 · MTES-MCT/monitorfish," Accessed: Sep. 30, 2023. [Online]. Available: https://github.com/MTES-MCT/monitorfish/pull/953/commits/1dc01c0d82261bf05277366d954fa5d912632091

[91] "react-customizable-progressbar," Accessed: Sep. 30, 2023. [Online]. Available: https://github.com/martyan/react-customizable-progressbar

[92] "fixed useEffect memory leak error by yoon-bbox · Pull Request #49 · BalloonBox-Inc/scrt-network-oracle-client," Accessed: Sep. 30, 2023. [Online]. Available: https://github.com/BalloonBox-Inc/scrt-network-oracle-client/pull/49/commits

[93] "website," Accessed: Oct. 1, 2023. [Online]. Available: https://github.com/momentum-mod/website

[94] "patternfly-react," Accessed: Oct. 1, 2023. [Online]. Available: https://github.com/patternfly/patternfly-react

[95] "fix: input refocused after blur by ThanoozN · Pull Request #541 · s-yadav/react-number-format," Accessed: Oct. 1, 2023. [Online]. Available: https://github.com/s-yadav/react-number-format/pull/541/files

[96] "Fix missing clear timeout by tjbo · Pull Request #731 · helpscout/hsds-react," Accessed: Oct. 1, 2023. [Online]. Available: https://github.com/helpscout/hsds-react/pull/731/files

[97] "Fix bugs and styling by edreichua · Pull Request #233 · dartmouth-cs98/project-dartmap," Accessed: Oct. 1, 2023. [Online]. Available: https://github.com/dartmouth-cs98/project-dartmap/pull/233/files

[98] "Vue-tree," Accessed: Jan. 13, 2024. [Online]. Available: https://github.com/wsfe/vue-tree
[99] "openMCT," Accessed: Jan. 1, 2024. [Online]. Available: https://github.com/nasa/openmct
[100] "CLRfund," Accessed: Jan. 1, 2024. [Online]. Available: https://github.com/ethereum/clrfund
[101] "web-mapviewer," Accessed: Jan. 1, 2024. [Online]. Available: https://github.com/geoadmin/web-mapviewer
[102] "InstaLog," Accessed: Jan. 1, 2024. [Online]. Available: https://github.com/GeekAbdou/InstaLog
[103] "n8n," Jan. 2024. Accessed: Jan. 1, 2024. [Online]. Available: https://github.com/n8n-io/n8n
[104] "UI," Accessed: Jan. 1, 2024. [Online]. Available: https://github.com/PrefectHQ/ui
[105] "pycontw-frontend," Accessed: Jan. 1, 2024. [Online]. Available: https://github.com/pycontw/pycontw-frontend
[106] "J. 2nfm, " Accessed: Jan. 1, 2024. [Online]. Available: https://github.com/codysherman/2nfm
[107] "docs," Accessed: Jan. 1, 2024. [Online]. Available: https://github.com/vuejs/docs
[108] H. B. Mann, "On a test of whether one of two random variables is stochastically larger than the other," *Ann. Math. Statist.*, vol. 18, no. 1, pp. 50–60, Mar. 1947.
[109] "AbortController - Web APIs | MDN," Accessed: Mar. 13, 2024. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/AbortController#browser_compatibility
[110] "ng-clarity," Accessed: Feb. 11, 2024. [Online]. Available: https://github.com/vmware-clarity/ng-clarity
[111] "octant," Accessed: Sep. 24, 2023. [Online]. Available: https://github.com/vmware-archive/octant
[112] "Patternfly-react," Accessed: Sep. 24, 2023. [Online]. Available: https://github.com/patternfly/patternfly-react
[113] "retail-UI," Accessed: Sep. 24, 2023. [Online]. Available: https://github.com/skbkontur/retail-ui
[114] Y. Tao, J. Kim, S. Kim, and C. Xu, "Automatically generated patches as debugging aids: a human study," in *Proc. 22nd ACM SIGSOFT Int. Symp. Found. Softw. Eng. (FSE)*, New York, NY, USA: ACM, 2014, pp. 64–74, doi: 10.1145/2635868.2635873.
[115] "IEEE xplore full-text pdf:," Accessed: Jul. 16, 2024. [Online]. Available: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6569740&tag=1