LEAKPAIR: Proactive Repairing of Memory Leaks in Single Page Web Applications

Arooba Shahoor
Kyungpook National University
Daegu, Republic of Korea
arooba.shahoor@knu.ac.kr

Askar Yeltayuly Khamit
UNIST
Ulsan, Republic of Korea
khamit.askar@unist.ac.kr

Jooyong Yi
UNIST
Ulsan, Republic of Korea
jooyong@unist.ac.kr

Dongsun Kim[†]
Kyungpook National University
Daegu, Republic of Korea
darkrsw@knu.ac.kr

***Abstract*—Modern web applications often resort to application development frameworks such as React, Vue.js, and Angular. While the frameworks facilitate the development of web applications with several useful components, they are inevitably vulnerable to unmanaged memory consumption since the frameworks often produce Single Page Applications (SPAs). Web applications can be alive for hours and days with behavior loops, in such cases, even a single memory leak in a SPA app can cause performance degradation on the client side. However, recent debugging techniques for web applications still focus on memory leak detection, which requires manual tasks and produces imprecise results.**

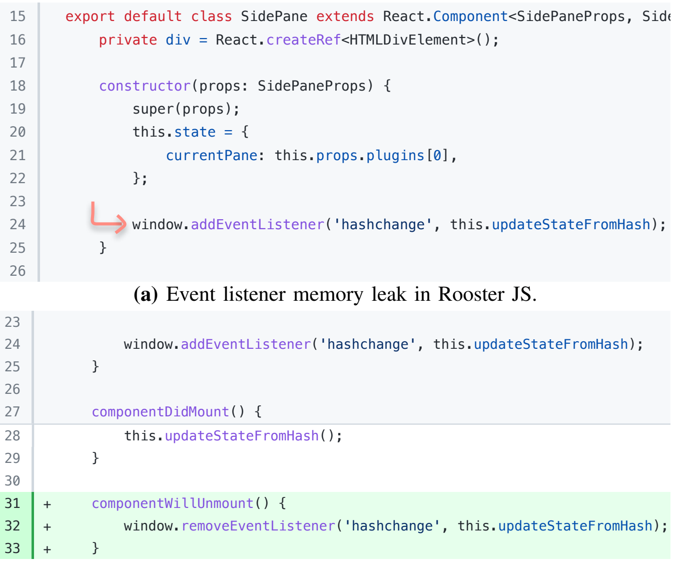
We propose LEAKPAIR, a technique to repair memory leaks in single page applications. Given the insight that memory leaks are mostly non-functional bugs and fixing them might not change the behavior of an application, the technique is designed to proactively generate patches to fix memory leaks, without leak detection, which is often heavy and tedious. 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 evaluate the technique on more than 20 open-source projects without using explicit leak detection. The patches generated by our technique are also submitted to the projects as pull requests. The results 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.

***Index Terms*—memory leaks, program repair, non-intrusive fixes, single page applications**

I. INTRODUCTION

Up until 2010, the website realm was mostly comprised of MPAs (Multiple Page Applications), where each page had to re-fetch and reload the entire webpage 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 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].



```
export default class SidePane extends React.Component<SidePaneProps, Sid
    private div = React.createRef<HTMLDivElement>();

    constructor(props: SidePaneProps) {
        super(props);
        this.state = {
            currentPane: this.props.plugins[0],
        };

        window.addEventListener('hashchange', this.updateStateFromHash);
    }

```

(a) Event listener memory leak in Rooster JS.

```

        window.addEventListener('hashchange', this.updateStateFromHash);
    }

    componentDidMount() {
        this.updateStateFromHash();
    }

+   componentWillUnmount() {
+       window.removeEventListener('hashchange', this.updateStateFromHash);
+   }
```

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

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

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 every object should reside in a single page. Therefore, SPAs inevitably rely on the garbage collectors of browsers to manage the memory space. Moreover, SPAs are highly likely to retain many loops (i.e., navigate back to the previous page) and the loops can rapidly add unnecessary objects that do not get garbage-collected due to unintentional reference. 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.

[†]Corresponding author.

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 Figure 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 global (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 (Figure 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) between 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:

- LEAKPAIR, a novel proactive approach to generating non-intrusive patches for fixing memory leaks in single page applications (SPAs).
- Four behavior-preserving fix patterns dedicated to repairing memory leaks in SPAs, which LEAKPAIR can leverage to generate non-intrusive patches.
- Empirical results of evaluating our approach on 37 open-source projects, which show the effectiveness and non-intrusiveness when repairing memory leaks in SPAs.

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 MPA, the actions taken by the user on the webpage trigger HTTP requests to the server; the server responds with a new page for each request, which means a page refresh for each interaction. In addition, the user session and data are persisted on the server; any time the session state or data is needed or updated, the server needs to be queried, and the client (and the user) needs to wait for the update to be completed on the server, resulting in a poor app responsiveness [3].

In contrast, SPA implements the majority of the logic for view generation on the Client side. A single `.html` file is loaded once, at the start of the program load, which is the only full browser load throughout the app. The single file contains multiple templates for different 'views', which are rendered on demand. Upon a user query, data is fetched from the server, and a template is updated with the data in real-time, without requiring a page reload. In addition, SPA caches all the received data from the server so that the user is still able to interact with the app in case of poor connection or connection loss, and any new data can be synced once the connection improves/restores [3].

In SPA, the job of merging data with views moves from the server to the client. The single HTML file (1) contains templates where data can be inserted and (2) generates a new 'view' that is equivalent to a new page in MPA. The logic of merging the data with the right template, routing to the right view, and maintaining the life cycle of a single view is accomplished via SPA frameworks such as Angular, React, Vue etc. When a user navigates to a "new page", the SPA framework is simply switching from one "view" to another [4].

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. 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 [18] revealed public-facing SPAs leaking up to 186 MB per interaction.

Furthermore, as will be demonstrated in the next section, since such leaks are hard to 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. Non-intrusive repair without replicating actual memory leaks

We figured out that it is challenging and non-trivial if a developer tries to diagnose memory leaks in SPAs. 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 [19]. Consider testimonials [20], [21], [22] 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 [23].*

> *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...[24].*

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.

Non-intrusive patches: Our intuition here is to apply non-intrusive patches [25] 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 [25] to fix performance bugs. However, ours is the first work using non-intrusive patches to fix memory leak issues, to the best of our knowledge.

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.[26], [13], [15]) 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.[27], [28]) also require a test suite to validate the generated patches. As will be shown in Section VI-A, the current neural program repair techniques such as CoCoNuT [28] are not capable of fixing the memory leaks of SPAs. The issue of the trustworthiness of the generated patches is also a concern for such techniques.

In comparison, we curate patch 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 [29] and SAVER [30], tools fixing the memory leaks of C/Java[*] 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. The event-driven / object-oriented nature of SPAs makes it easy for the developers to assess the correctness of the patches — a patch is often applied to an object destructor, which is called implicitly when an event occurs. 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.

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.

[*] SAVER cannot handle Java programs.

B. Mining fix patterns for SPA memory leaks

Since our goal is to identify recurring and common patterns of memory leaks and their corresponding patches in SPAs, we first collect the most common leaks available publicly by using specific keywords such as 'leak' and 'React'. Our search targets were `GitHub.com` and `stackoverflow.com`. 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 [16], [31], [25], [32], [33], [14], [15] have demonstrated that this strategy is effective and useful, as we can reuse the fix patterns many times once they have been identified.

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 keywords such as 'leak', 'memory usage' or 'memory leak' and that explicitly pertain to JavaScript applications, (2) For GitHub, we search through 1,000 commits, PRs, issues, and discussions containing any of the above keywords, along with the labels 'React' or 'Angular' (being the most commonly used frameworks for SPA development).

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 (4) four 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 [34].

C. Fix patterns

As already discussed in Section 2.2, 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 four 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) [35]. 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 [36], 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 `observer2` (see pseudo-code below) as input to the `takeUntil()` operator, and in the destructor we make `observer2` emit a value (using the `next()` and `complete()` methods), that will clear the subscription and thus prevent the memory leak[†].

```
- observer1.subscribe(() => {...})
+ observer1.pipe(takeUntil(observer2)).subscribe(() =>
     {...})
...
+ destructorMethodDeclaration() {
       ...
+      observer2.next()
+      observer2.complete()
+ }
```

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 `listenerHandler` (see pseudo-code below) and passing it to

[†] https://github.com/blackbaud/skyux/pull/376/files

`removeEventListener()` method[‡].

```
function listenerHandler() {
        ...
   }
...
eventTarget.addEventListener(eventType, listenerHandler ,
    options)
...
+ destructorMethodDeclaration() {
          ...
      +     eventTarget.removeEventListener(eventType,
        listenerHandler, options)
      + }
```

FP3a. Uncleared Timers: `setTimeOut`. 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[§].

```
- setTimeOut(() => {...})
+ timeOutID = setTimeOut(() => {...})
...
+ destructorMethodDeclaration() {
      ...
+     clearTimeOut(timeOutID)
+ }
```

FP3b. Uncleared Timers: `setInterval`. 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 [37]. 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[¶].

```
- setInterval(() => {...})
+ intervalID = setInterval(() => {...})
...
+ destructorMethodDeclaration() {
      ...
+     clearInterval(intervalID)
+ }
```

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 [38]. 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[‖].

```
- requestAnimationFrame(() => {...})
+ let requestID = requestAnimationFrame(() => {...})

+ destructorMethodDeclaration() {
      ...
+     cancelAnimationFrame(requestID)
+ }
```

1) Edit script: 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], [39], [15], [25]. Therefore, we implement the scripts for our tool, which are available in our replication package [40].

2) Coverage of the patterns: The four fix patterns cover most of the fixed memory leaks we have examined. Following the procedures described in Section III-B, we identified 124 and 65 memory leak bugs in SPAs based on React and Angular, respectively, as a result. These bugs have been confirmed and fixed by the developers of the SPAs. Our four fix patterns can fix 102 out of 124 (82.2%) and 57 out of 65 (87.6%) already-known React and Angular-related memory leak bugs, respectively. The distribution of fix patterns (FP1–FP4) for React leak bugs is 2/124 (1.6%), 37/124 (29.8%), 45/124 (36.3%), and 18/124 (14.5%), resepectively. The distribution for Angular is 47/65 (72.3%), 7/65 (10.8%), 2/65 (3.1%), and 1/65 (1.5%), respectively. The full list of known memory leak bugs examined is available in our replication package [40].

[‡] https://github.com/microsoft/roosterjs/pull/921/files

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

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

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

D. 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-C1 are executed for each file. Specifically, it follows the following procedure.

- **Parsing and Detecting:** LeakPair makes use of the Babel compiler [41] in conjunction with Facebook's *jscodeshift* [42] 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 and React components (Vue is not supported currently) by matching their syntax definition. Once a component from these frameworks is identified, it detects whether the component implements any of the four 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 [43] 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) as well as 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).

E. Non-intrusive patch generation

As LeakPair aims at proactively generating non-intrusive patches for memory leaks in SPAs, we apply the following procedures 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. Our approach carefully scans the target SPA once again to figure out whether there is any cleanup code for the specific object for each pattern. LeakPair generates a patch by using the pattern only when there is no cleanup code to avoid redundant patches, which can unnecessarily bloat the source code.

TABLE I: Subjects with unknown memory leaks.

ID	Program	Type	SPA Framework	Commit Hash
U1	react-zoom-pan-pinch [44]	Library	React	fdc030
U2	Angular Extentions Elements [45]	Library	Angular	d9a4e4
U3	Evergreen [46]	Framework	React	82c3a8
U4	ngx-datatable [47]	Library	Angular	6184c9
U5	react-multi-carousel [48]	Library	React	525793
U6	codetekt (Frontend) [49]	Website	Angular	7b8289
U7	skbkontur/retail-ui [50]	Framework	React	32f3cf
U8	Aam Digital [51]	Web app	Angular	304ff9
U9	Replay's DevTools [52]	Library	React	24d10f
U10	ngx-bootstrap [53]	Framework	Angular	663c70

The full list of subjects used for our experiment is available in the replication package [40].

TABLE II: Subjects with known memory leaks.

ID	Program	Type	SPA Framework	Commit Hash
K1	react-zoom-pan-pinch [44]	Library	React	6e35b3
K2	Fundamental Library for Angular [54]	Library	Angular	be9629
K3	react-multi-carousel [48]	Library	React	5d252d
K4	Angular Components [45]	Framework	Angular	1bbb29
K5	Material UI [55]	Framework	React	e92b1c
K6	Angular Components documentation [56]	Website	Angular	e8cb0d
K7	Rooster [6]	Library	React	c3f2f0
K8	Octant [57]	Framework	Angular	b079ad
K9	Evergreen [46]	Framework	React	a716f4
K10	Transloco [58]	Library	Angular	2338a0

The full list of subjects used for our experiment is available in the replication package [40].

- **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?

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.

B. Experiement 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, 37 projects are selected as the subjects for our experiments to assess LEAKPAIR; 19 projects have unknown new memory leaks while 18 projects out of them have already known memory leaks. In this paper, we focus on and report only on the results of 10 unknown and 10 known subjects due to space limitations. Tables I and II list 10 unknown and 10 known subjects, respectively, out of our 37 subjects; the complete results of experiments for the entire set of subjects are available in our replication package [40].

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 [34], 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 [40].

To compare the memory consumption, we compute the memory footprints before and after applying LEAKPAIR. For each subject, the corresponding scenario file is executed 10 times with `loop=10` (i.e., 10 × 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 [34], 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 [59], 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.

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 III. We apply the tool to each subject listed in Tables I (projects with unknown memory leaks) and II (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 size of the heap in each subject before and after applying our tool, and the difference.

As shown in Tables IV and III, respectively, LEAKPAIR can reduce both memory consumption and potentially leaking objects. 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. The

TABLE III: Memory consumption results before and after applying LeakPair to the subjects in Table I.

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	4 clusters	3 clusters	1 cluster	47.9 MB	43.7 MB	5.2 MB (10.8%)∗∗
U2	FP1, FP2	13 clusters	10 clusters	3 clusters	17.4 MB	16.7 MB	0.7 MB (4%)∗∗
U3	FP4	5 clusters	3 clusters	2 clusters	35.5 MB	29.2 MB	6.3 MB (17.7%)∗∗
U4	FP3, FP4	8 clusters	7 clusters	1 cluster	83.9 MB	66.9 MB	15.0 MB (17.8%)∗∗
U5	FP3	5 clusters	4 clusters	1 cluster	27.9 MB	23.5 MB	3.4 MB (12.1%)∗
U6	FP1, FP3	12 clusters	9 clusters	3 clusters	65.1 MB	63.6 MB	1.5 MB (2.3%)∗∗
U7	FP3	7 clusters	5 clusters	2 clusters	105 MB	100 MB	5.0 MB (4.7%)∗
U8	FP1	2 clusters (95,352 objects)	2 clusters (73,951 objects)	20,000+ objects	318.9 MB	264.3 MB	57.6 MB (18%)∗∗
U9	FP3	5 clusters	3 clusters	2 clusters	27 MB	26.4 MB	0.6 MB (2.2%)
U10	FP1	6 clusters	4 clusters	2 clusters	101.7 MB	99.7 MB	2.0 MB (1.9%)∗∗

∗: p-value $<$ 0.05, ∗∗: p-value $<$ 0.01. The full list of subjects used for our experiment is available in the replication package [40].

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	Heapsize After applying LeakPair	Total Heap Size Reduction
K1	FP2	3.9 clusters	3.1 clusters	0.8 clusters	28.19 MB	28.09 MB	0.1 MB (0.3%)∗
K2	FP1	1 cluster (70.7 objects)	1 cluster (66.9 objects)	3.8 objects	53.4 MB	52.1 MB	1.3 MB (2.4%)∗∗
K3	FP3	5 clusters	4 cluster	1 cluster	17.19 MB	16.87 MB	0.32 MB (1.8%)
K4	FP1	5.6 clusters	5.3 cluster	0.3 cluster	13.08 MB	12.98 MB	0.1 MB (0.7%)∗
K5	FP3	13.8 clusters	13.7 clusters	0.1 clusters	13 MB	12.9 MB	0.1 MB (0.7%)
K6	FP1	2 clusters (245.6 objects)	2 clusters (154.5 objects)	91.1 objects	11.8 MB	11.8 MB	0.0 MB (0%)
K7	FP2	4 clusters (4370.8 objects)	4 clusters (4295.2 objects)	75.6 objects	10.9 MB	10.87 MB	0.03 MB (0.2%)
K8	FP1	16.4 clusters	15.6 clusters	0.8 clusters	61.08 MB	60.88 MB	0.9 MB (1.4%)
K9	FP3	4 clusters (3923.9 objects)	4 clusters (3662.7 objects)	261.2 objects	27.5 MB	27.3 MB	0.2 MB (0.7%)∗
K10	FP1	1 cluster	0 cluster	1 cluster	9.2 MB	9.2 MB	0.0 MB (0%)

∗: p-value $<$ 0.05, ∗∗: p-value $<$ 0.01. The full list of subjects used for our experiment is available in the replication package [40].

TABLE V: Results of pull requests reporting the patches generated by LeakPair, which fix unknown leaks in subjects listed in Table I.

Agreed			Disagreed	Ignored	Total
Merged	Approved	Improved			
9	2	1	0	8	20

higher effectiveness shown in Table III might indicate that the subjects with unknown leaks paid less attention to memory management while those in Table IV paid more attention; which is why we could identify already-known memory leak patches from the subjects.

The patches do not introduce any new leaks, as shown in the plots in Figure 2. The plots illustrate the changes in the memory heap size in one execution for each subject. 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 to reducing memory consumption, or at least, they do not add to it. One of the patches (`K9`) reduces the heap size even more than the developer's patch.

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 the 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.*

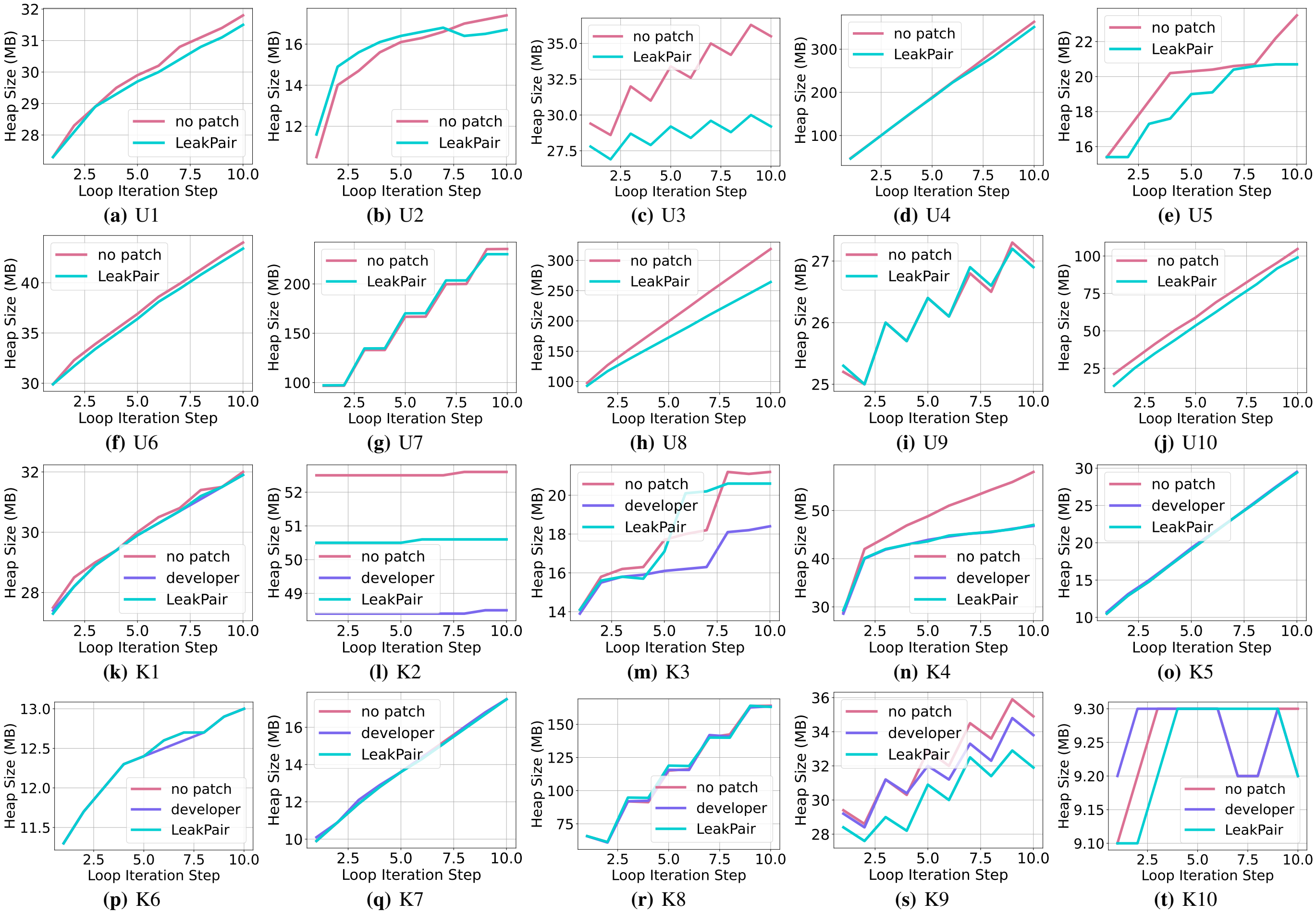
B. RQ2: Are the patches by LeakPair *acceptable?*

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 I, and observing the outcome of pull requests. We submitted 20 pull requests for 17 of those subjects 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 [34].

Table V contains the results of the live study up to the date of the submission. 11 out of 20 PRs (60%) 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 in the near future.

One of our PRs inspired the project owner to fix a similar memory leak pattern together with 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. Eight PRs did not get any response from the developers up to the date of submission.



(a) U1 **(b)** U2 **(c)** U3 **(d)** U4 **(e)** U5

(f) U6 **(g)** U7 **(h)** U8 **(i)** U9 **(j)** U10

(k) K1 **(l)** K2 **(m)** K3 **(n)** K4 **(o)** K5

(p) K6 **(q)** K7 **(r)** K8 **(s)** K9 **(t)** K10

Fig. 2: Heap size over loops after applying LEAKPAIR to the subjects listed in Tables III and IV.

TABLE VI: Test execution applying LEAKPAIR to the subjects in Table I.

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	41.623 s	43.683 s
U9	43 passed of 43	43 passed of 43	6.478 s	6.318 s
U10	12 passed of 12	12 passed of 12	0.3 s	0.3 s

The full list of subjects used for our experiment is available in the replication package [40].

> **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?

To show the non-intrusiveness of the patches generated by our tool, 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 I and II, respectively. The tables report on the execution time of the test suites as well.

TABLE VII: Test execution results 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
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

The full list of subjects used for our experiment is available in the replication package [40].

As shown in Tables VI and VII, the patches generated by LEAKPAIR do not introduce any new positive or negative test outcomes. For subjects with some skipped and failed test cases, we checked if any new positive or negative test cases had replaced the previous outcomes. As a result, we found no replacement, which indicates 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.

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.*

VI. Discussion

A. Comparison against state-of-the-art

To the best of our knowledge, LEAKPAIR is the only program repair tool that fixes memory leaks in JavaScript programs. As of February 2023, the only alternative program repair tool that can deal with Javascript programs is COCONUT [28]. COCONUT is a recent learning-based program repair tool. Since its training data contains Javascript programs (3,217,093 programs obtained from 10,163 open-source projects), it can be used to fix Javascript programs. As a general-purpose APR tool, COCONUT can also be used to fix memory leaks.

We assess the performance of COCONUT in fixing memory leaks by applying it to all pairs of buggy and fixed versions from which we mined our fix patterns. COCONUT, like most program repair tools, requires buggy lines, which we provide with ground truth patches. In case the ground-truth fix modifies multiple lines, we apply COCONUT to each of those lines. Then, at each buggy line, we compare the ground-truth fix and the 1,000 COCONUT-generated fixes (COCONUT uses beam search with a beam width 1,000). In none of the buggy lines, COCONUT generates a ground-truth fix.

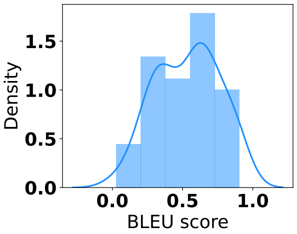


Fig. 3: Distribution of the BLEU scores of COCONUT-generated patches.

To assess COCONUT at a finer-granularity level (token level), we compute BLEU scores at each line. Figure 3 shows the distribution of the obtained BLEU scores. At each line, we consider only the maximum score out of 1,000.

Figure 4 shows the patches generated by COCONUT and LEAKPAIR, as well as the developer patch for the react-zoom-pan-pinch project**. While the COCONUT-generated patch appears to be similar to the developer patch with the BLEU score of 0.62, it is not even syntactically correct. This issue is typical of learning-based program repair tools. Meanwhile, LEAKPAIR successfully generates a patch semantically equivalent to the developer patch using the fix pattern FP2. Although the fix statement is added in a different location than in the developer patch, it is still semantically equivalent to the developer patch, since both patches remove the same event listener of the same event type.

```
  window.removeEventListener("mouseup", this.onPanningStop,
     ↪ passive);
+ document.removeEventListener( ’keydown’, this.
     ↪ setKeyPressed , ;
  window.removeEventListener("keyup", this.setKeyUnPressed,
     ↪ passive);
  ...
  handleCancelAnimation(this);
```

(a) A patch generated by COCONUT.

```
  window.removeEventListener("mouseup", this.onPanningStop,
     ↪ passive);
  window.removeEventListener("keyup", this.setKeyUnPressed,
     ↪ passive);
  ...
  handleCancelAnimation(this);
+ document.removeEventListener("mouseleave", this.
     ↪ clearPanning, passive);
```

(b) A patch generated by LEAKPAIR.

```
  window.removeEventListener("mouseup", this.onPanningStop,
     ↪ passive);
+ document.removeEventListener("mouseleave", this.
     ↪ clearPanning, passive);
  window.removeEventListener("keyup", this.setKeyUnPressed,
     ↪ passive);
  ...
  handleCancelAnimation(this);
```

(c) The developer patch.

Fig. 4: Patches for the react-zoom-pan-pinch project (↪ indicates the beginning of the wrapped lines).

Our COCONUT-experiment results suggest that (1) the learning-based approach such as COCONUT shows the potential to generate a correct fix (the highest BLEU score is 0.90), but (2) the performance is not as good as being able to generate a correct fix (not a single version is correctly fixed). While general-purpose program repair is attractive, it is not a panacea. For bugs that can be fixed with patterns, such as memory leaks, our pattern-based approach works better and more reliably.

B. 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.

Threats to internal validity may include fix patterns manually extracted by the authors. To address this threat, each fix pattern is supported by real patches that fix memory leaks in SPAs implemented by React and Angular.

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

[**] https://github.com/prc5/react-zoom-pan-pinch/pull/270/commits/6e35b3a552c7780aa0cef944f37a9d60d904a3c3

the correctness of the behavior in the applications, it might be enough to preserve major functionalities in the applications from the maintenance perspective.

VII. Related Work

A. Pattern-based Program Repair

Program repair with fix patterns (or fix templates) was first introduced by Kim et al. [16], where the authors manually inspected 60,000+ human-written patches. Based on the inspection, common patterns were derived that were then implemented as automated fix templates in PAR (Patch-Based Automated Program Repair). The tool was evaluated by applying it to 119 real-world bugs and comparing the number of patches generated by PAR that were approved by 253 human subjects with those generated by GenProg [26]. Patches generated by PAR were shown to have a higher acceptance ratio.

Pattern-based program repair has been improved and leveraged with many other ideas. There are studies that extract patterns for different targets such as JavaScript faults [31] and performance bugs [25]. Researchers leveraged diverse sources of fix patterns, for example, Q&A posts [32], similar snippets [33], fault localization results [14], and static analysis warnings [39]. In addition, TBar [15] incorporated common fix patterns from other existing studies and showed that fix patterns are effective when fixing bugs.

Fix patterns are also utilized to generate non-intrusive patches as well. The authors of Caramel [25] examined patches submitted to fix performance bugs in open-source projects written in C and Java languages. The technique identifies potential performance issues in a program. and generates patches that do not change the functionality of the program, i.e., non-intrusive fixes.

B. Memory Leak Debugging

There have been some techniques proposed to address memory leaks in Javascript projects. Qian et al. [10] proposed a technique that reports the suspected leaking objects by collecting the application heap snapshots and using a lightweight statistical algorithm that combines several heuristics. BLeak [12] 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 state should consume the same amount of memory; therefore, if there is sustained growth in memory consumption between the loops to the same state, it is a valid indicator of memory leakage.

Another common line of research involves dynamic approaches. One dynamic approach for non-garbage-collected languages, presented by Azhari et al. [8] performs memory leak detection by memory block growth analysis. Another dynamic leak detector, DEF_LEAK [60], employs symbolic execution to detect memory leaks across all paths of execution. LeakSpot [11] addresses memory leaks in JavaScript by leveraging a heap snapshot model. MemInsight [9] instruments the JavaScript code and provides a detailed analysis of the applications' memory dynamics. Memory Validator [61] is a popular memory leak and memory errors detection tool for C, C++, C#, Visual Basic and Fortran.

Recently, memory leak detection techniques have leveraged neural networks. MVD [62] makes use of a novel kind of graph neural network called flow- sensitive graph neural network (FS-GNN). FS-GNN helps capture critical contextual data of the code by embedding both statements and flow information in order to learn program semantics. This model can be trained to learn vulnerability patterns from the source code as well as detect statements that are suspected to be vulnerable. It does so by incorporating semantic information such as call relations and return values from Call graphs into the basic Program Dependence Graph (PDG).

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.

IX. Data Availability

We make the replication package publicly available, which includes all the code and datasets to reproduce our experiments at https://figshare.com/s/5991a6f89800906176a2 [40].

Acknowledgments

This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (No. 2021R1A5A1021944 and 2021R1I1A3048013) and the Institute for Information & Communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (No.2021-0-01001). Additionally, the research was supported by Kyungpook National University Research Fund, 2020.

References

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

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

[3] G. Fink, I. Flatow, and SELA 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?” [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,” [Online; accessed 15. Feb. 2023]. [Online]. Available: https://auth0.com/blog/four-types-of-leaks-in-your-javascript-code-and-how-to-get-rid-of-them

[6] “roosterjs,” [Online; accessed 15. Feb. 2023]. [Online]. Available: https://github.com/microsoft/roosterjs

[7] “Window: hashchange event - web apis | mdn,” [Online; accessed 15. Feb. 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 *2021 IEEE International Conference on Big Data (Big Data)*. Orlando, FL, USA: IEEE, 2021, pp. 3020–3030.

[9] S. H. Jensen, M. Sridharan, K. Sen, and S. Chandra, “Meminsight: Platform-independent memory debugging for javascript,” in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2015. New York, NY, USA: Association for Computing Machinery, 2015, p. 345–356. [Online]. Available: https://doi.org/10.1145/2786805.2786860

[10] J. Qian, L. Wang, and X. Zhou, “A lightweight approach to detect memory leaks in javascript (s),” in *International Conference on Software Engineering and Knowledge Engineering*. San Francisco, California, USA: KSI Research 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, p. 97–123, jan 2017. [Online]. Available: https://doi.org/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, p. 146–153, oct 2020. [Online]. Available: https://doi.org/10.1145/3422598

[13] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra, “SemFix: Program Repair via Semantic Analysis,” in *Proceedings of the 2013 International Conference on Software Engineering*, ser. ICSE ’13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 772–781.

[14] A. Koyuncu, K. Liu, T. F. Bissyandé, D. Kim, J. Klein, M. Monperrus, and Y. Le Traon, “Fixminer: Mining relevant fix patterns for automated program repair,” *Empirical Softw. Engg.*, vol. 25, no. 3, p. 1980–2024, may 2020. [Online]. Available: https://doi.org/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 *ISSTA 2019: Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2019. New York, NY, USA: Association for Computing Machinery, 2019, p. 31–42. [Online]. Available: https://doi.org/10.1145/3293882.3330577

[16] D. Kim, J. Nam, J. Song, and S. Kim, “Automatic patch generation learned from human-written patches,” in *2013 35th International Conference on Software Engineering (ICSE)*. San Francisco, CA, USA: IEEE, 2013, pp. 802–811.

[17] C. L. Goues, M. Pradel, and A. Roychoudhury, “Automated program repair,” *Communications of the ACM*, vol. 62, no. 12, pp. 56–65, Nov. 2019.

[18] “Introducing fuite: a tool for finding memory leaks in web apps,” [Online; accessed 15. Feb. 2023]. [Online]. Available: https://nolanlawson.com/2021/12/17/introducing-fuite-a-tool-for-finding-memory-leaks-in-web-apps

[19] “A tour of V8: Garbage Collection,” [Online; accessed 15. Feb. 2023]. [Online]. Available: https://jayconrod.com/posts/55/a-tour-of-v8--garbage-collection

[20] “pomodore-discord-bot,” [Online; accessed 15. Feb. 2023]. [Online]. Available: https://github.com/MarcoPereira27/pomodore-discord-bot/issues/4

[21] “Strange nodejs memory leak,” [Online; accessed 15. Feb. 2023]. [Online]. Available: https://stackoverflow.com/questions/63661738/strange-nodejs-memory-leak

[22] “angular,” [Online; accessed 15. Feb. 2023]. [Online]. Available: https://github.com/angular/angular/issues/27803

[23] “Solving Memory Leaks in Large React Application,” online; accessed 15. Feb. 2023. [Online]. Available: https://stackoverflow.com/questions/63813604/solving-memory-leaks-in-large-react-application

[24] “angular,” [Online; accessed 15. Feb. 2023]. [Online]. Available: https://github.com/angular/angular/issues/20007

[25] A. Nistor, P.-C. Chang, C. Radoi, and S. Lu, “Caramel: Detecting and Fixing Performance Problems That Have Non-intrusive Fixes,” in *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ser. ICSE ’15. Piscataway, NJ, USA: IEEE Press, 2015, pp. 902–912.

[26] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest, “Automatically finding patches using genetic programming,” in *Proceedings of the 31st International Conference on Software Engineering*, ser. ICSE ’09. IEEE Computer Society, 2009, pp. 364–374.

[27] Q. Zhu, Z. Sun, Y.-a. Xiao, W. Zhang, K. Yuan, Y. Xiong, and L. Zhang, “A syntax-guided edit decoder for neural program repair,” in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. New York, NY, USA: Association for Computing Machinery, Aug. 2021, pp. 341–353.

[28] 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 *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. Virtual Event USA: ACM, Jul. 2020, pp. 101–114. [Online]. Available: https://dl.acm.org/doi/10.1145/3395363.3397369

[29] R. van Tonder and C. L. Goues, “Static automated program repair for heap properties,” in *Proceedings of the 40th International Conference on Software Engineering*, 2018, pp. 151–162.

[30] S. Hong, J. Lee, J. Lee, and H. Oh, “SAVER: scalable, precise, and safe memory-error repair,” in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 271–283.

[31] F. S. Ocariza, Jr., K. Pattabiraman, and A. Mesbah, “Vejovis: Suggesting fixes for javascript faults,” in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014. New York, NY, USA: Association for Computing Machinery, 2014, p. 837–847. [Online]. Available: https://doi.org/10.1145/2568225.2568257

[32] X. Liu and H. Zhong, “Mining stackoverflow for program repair,” in *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. Campobasso, Italy: IEEE, 2018, pp. 118–129.

[33] J. Jiang, Y. Xiong, H. Zhang, Q. Gao, and X. Chen, “Shaping program repair space with existing patches and similar code,” in *ISSTA 2018: Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2018. New York, NY, USA: Association for Computing Machinery, 2018, p. 298–309. [Online]. Available: https://doi.org/10.1145/3213846.3213871

[34] G. C. Liang Gong, “MemLab: An open source framework for finding JavaScript memory leaks,” *Engineering at Meta*, vol. , no. , Oct. 2022. [Online]. Available: https://engineering.fb.com/2022/09/12/open-source/memlab

[35] “RxJS - Observable,” [Online; accessed 16. Feb. 2023]. [Online]. Available: https://rxjs.dev/guide/observable

[36] “RxJS - takeUntil,” [Online; accessed 16. Feb. 2023]. [Online]. Available: https://rxjs.dev/api/operators/takeUntil

[37] “Codecademy,” [Online; accessed 17. Feb. 2023]. [Online]. Available: https://www.codecademy.com/courses/react-101/lessons/component-lifecycle-methods/exercises/componentwillunmount

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

[39] K. Liu, A. Koyuncu, D. Kim, and T. F. Bissyande, “Avatar: Fixing semantic bugs with fix patterns of static analysis violations,” in *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. Los Alamitos, CA, USA: IEEE Computer Society, feb 2019, pp. 1–12. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/SANER.2019.8667970

[40] “LeakPair: An automated memory leak repair tool for Single Page Applications,” [Online; accessed 17. Feb. 2023]. [Online]. Available: https://figshare.com/s/5991a6f89800906176a2

[41] “Babel · The compiler for next generation JavaScript,” [Online; accessed 17. Feb. 2023]. [Online]. Available: https://babeljs.io

[42] “jscodeshift,” [Online; accessed 16. Feb. 2023]. [Online]. Available: https://github.com/facebook/jscodeshift

[43] “recast,” [Online; accessed 16. Feb. 2023]. [Online]. Available: https://github.com/benjamn/recast

[44] “react-zoom-pan-pinch,” [Online; accessed 15. Feb. 2023]. [Online]. Available: https://github.com/prc5/react-zoom-pan-pinch/pull/270/commits

[45] “angular-components,” [Online; accessed 16. Feb. 2023]. [Online]. Available: https://github.com/angular/components

[46] “evergreen,” [Online; accessed 16. Feb. 2023]. [Online]. Available: https://github.com/segmentio/evergreen

[47] “ngx-datatable,” [Online; accessed 16. Feb. 2023]. [Online]. Available: https://github.com/swimlane/ngx-datatable

[48] “react-multi-carousel,” [Online; accessed 16. Feb. 2023]. [Online]. Available: https://github.com/YIZHUANG/react-multi-carousel

[49] “angular-ui,” [Online; accessed 16. Feb. 2023]. [Online]. Available: https://github.com/DetektivKollektiv/angular-ui

[50] “retail-ui,” [Online; accessed 16. Feb. 2023]. [Online]. Available: https://github.com/skbkontur/retail-ui/tree/retail-ui%401.11.1

[51] “ndb-core,” [Online; accessed 16. Feb. 2023]. [Online]. Available: https://github.com/Aam-Digital/ndb-core

[52] “devtools,” [Online; accessed 16. Feb. 2023]. [Online]. Available: https://github.com/replayio/devtools

[53] “ngx-bootstrap,” [Online; accessed 16. Feb. 2023]. [Online]. Available: https://github.com/valor-software/ngx-bootstrap

[54] “fundamental-ngx,” [Online; accessed 16. Feb. 2023]. [Online]. Available: https://github.com/SAP/fundamental-ngx

[55] “material-ui,” [Online; accessed 16. Feb. 2023]. [Online]. Available: https://github.com/mui/material-ui

[56] “material.angular.io,” [Online; accessed 16. Feb. 2023]. [Online]. Available: https://github.com/angular/material.angular.io

[57] “octant,” [Online; accessed 16. Feb. 2023]. [Online]. Available: https://github.com/vmware-archive/octant

[58] “transloco,” [Online; accessed 16. Feb. 2023]. [Online]. Available: https://github.com/ngneat/transloco/pull/65/files

[59] H. B. Mann, “On a Test of Whether one of Two Random Variables is Stochastically Larger than the Other,” *The Annals of Mathematical Statistics*, vol. 18, no. 1, pp. 50–60, Mar. 1947.

[60] B. Yu, C. Tian, N. Zhang, Z. Duan, and H. Du, “A dynamic approach to detecting, eliminating and fixing memory leaks,” *J. Comb. Optim.*, vol. 42, no. 3, pp. 409–426, Oct. 2021.

[61] “MemoryValidatorOverview,” [Online; accessed 17. Feb. 2023]. [Online]. Available: https://www.softwareverify.com/product/memory-validator

[62] S. Cao, X. Sun, L. Bo, R. Wu, B. Li, and C. Tao, “Mvd: Memory-related vulnerability detection based on flow-sensitive graph neural networks,” in *ICSE ’22: Proceedings of the 44th International Conference on Software Engineering*, ser. ICSE ’22. New York, NY, USA: Association for Computing Machinery, 2022, p. 1456–1468. [Online]. Available: https://doi.org/10.1145/3510003.3510219