# MemRed: Towards Reliable Web Applications

Masoomeh Rudafshani
University of Waterloo
mrudafsh@uwaterloo.ca

Paul A.S. Ward
University of Waterloo
pasward@uwaterloo.ca

Bernard Wong
University of Waterloo
bernard@uwaterloo.ca

## ABSTRACT

Current approaches for improving the reliability of web services focus on server side data collection and analysis to detect errors and prevent failures. However, significant portions of modern web applications are executed on the client browser with the server only acting as a data store. These applications are mostly developed using Javascript, which presents a challenge for developing reliable web applications due to a current lack of tools for debugging Javascript applications. In addition, these applications use AJAX to communicate with the server asynchronously; therefore they remain on the same page during their lifetime that can lead to runaway memory usage from even minor memory leaks. In this paper, we introduce MemRed, a system that improves the reliability of the client side of web applications. It achieves this goal by taking advantage of browser APIs to monitor web applications. It analyzes the collected data to detect excessive memory utilization and applies recovery action to hide failures from end users, if needed. Our prototype is implemented as an extension for the Chrome browser. The evaluation shows the effectiveness of recovery actions in lowering memory usage of web applications.

## Categories and Subject Descriptors

C.4 [**Performance of Systems**]: Fault-tolerance, Measurement techniques; B.8 [**Metrics**]: Performance Measures

## General Terms

Reliability, Performance, Measurement, Experiment

## Keywords

Monitoring, Error Detection, Web Application, Recovery

## 1. INTRODUCTION

The use of asynchronous communication techniques, such as AJAX, in modern web applications has enabled the development of Rich Internet Applications (RIAs) that provide similar functionality and interactivity as desktop applications. In these applications, large parts of the code are written in Javascript and run on the client browser, which provides a platform for their development and execution. However, it is hard to develop reliable web applications in Javascript since it is a weakly typed language [21] and there is a current lack of tools for debugging Javascript applications. Moreover, these applications often remain on the same page for hours without requiring a full page refresh since they asynchronously communicate with the server to receive updates on specific objects on the page. This can lead to excessive memory usage due to memory leaks or fragmentation. The growth in the size and complexity of the client side components in web applications, the potential for error accumulation due to long-lived web applications, and the weakness of Javascript in building large-scale reliable applications have motivated us to work on techniques to improve web application reliability.

Most current work focus on improving the reliability on the server-side [12, 17, 25, 14, 9, 16, 11]. However, we need new solutions on the client side as the differences between the client and server environments introduce many new reliability challenges. First, in a client browser, there are several web applications running simultaneously and errors in one web application may have some effects on the other applications. Second, approaches on the server side are meant to help system administrators who have more technical expertise compared to the ordinary user on the client side. Third, resources are limited on the client side which amplify failure perception by the user. It also limits the recovery mechanisms due to overhead associated with monitoring and recovery mechanisms. Finally, high interactivity on the client-side of web applications makes transparent recovery challenging. In addition, it makes user-perceived response time a priority compared to the throughput. Therefore, we need a new solution to make the client side of web applications more dependable.

In this paper, we present MemRed, a system for improving the reliability of web applications at runtime. MemRed is implemented as an extension inside the Chrome browser. It is designed to work by monitoring the memory usage of web applications and then analyzing the collected data to detect excessive memory usage, which could indicate a memory leak. If the data analysis predicts a failure, it tries to apply a recovery action at an appropriate time to hide the failures from the user.

Although the Chrome browser [1] provides high reliability by executing each web application within a separate process [23], there are several limitations to this approach. First, the Chrome loads a page made of several iframes into the same process so that objects within different iframes are able to refer to each other; therefore, iframes are not isolated [26]. Second, there is a limitation on the number of processes that Chrome will create, which depends on the available system resources. Upon reaching this limit, new pages share a process with currently opened pages. Third, an application within a separate process may suffer from errors or failures due to a bug in the code and process separation does not help in such scenarios. Therefore, we need new mechanisms for improving the reliability and availability of the client side of web applications. It is worth mentioning that although we are focusing on Javascript-based applications in the browser, the approach proposed is useful in other contexts as well. For example, Javascript is also used for application development in some operating systems such as Windows 8, along with CSS and HTML5. Therefore, our technique can be used for failure recovery of such applications.

Overall, in this paper we present our system for online detection and recovery of failures in web applications, and make three contributions. First, we provide evidence supporting the need for mechanisms to improve dependability on the client side of web applications. Second, we present a prototype of our approach for achieving this goal. Third, we show the results of an empirical study on a real-world web application as well as a benchmark application that demonstrate the effectiveness of MemRed's recovery actions.

## 2. RELATED WORK

Web applications are subjected to different types of failures. Functional errors in web applications have been studied by Ocariza et al. [21]. They categorize errors in Javascript-based web applications and study the correlation between application properties and failure frequencies. Failures due to security bugs have been studied in the past [15]. we focus on software aging [13] bugs which degrade software systems gradually, resulting in poor response time or crash. Specifically, we focus on excessive usage of memory in web applications.

To predict failures, Grottke et al.[14] make a model of system behaviour while the server is under artificial workload and then use this model during runtime to predict system failures. Alonso et al. [9] estimate time-to-failure of a web server based on the failure data. These approaches are not appropriate on the client-side since many applications are running inside the browser. Software rejuvenation [16, 11] has been used to recover from errors due to software aging and inspired us to have similar recovery actions for web applications.

To monitor web applications, Ajaxscope [18] acts as a proxy, instrumenting Javascript code before being received by the client. It receives data about application states, analyzes them, and notifies the server in case of an error. Richard et al. [24] study behaviour of web applications by instrumenting Javascript engine of Safari. To diagnose errors, Mugshot [20] and WaRR [10] record events on a Javascript page and replay them after failure to find the faults. While these techniques are useful in development phase of web applications, our focus is on runtime.

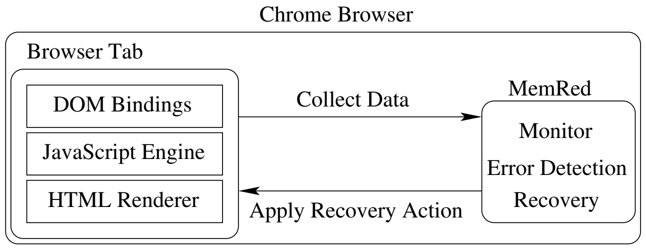

**Figure 1: Overall structure of MemRed running as an extension inside Chrome browser.**

## 3. IMPLEMENTATION

MemRed is implemented as an extension to the Chrome browser. We selected the Chrome browser as our system platform due to its high usage share [5]. MemRed, as shown in Figure 1, monitors the web pages in a browser, analyzes the collected data, and applies proactive recovery actions if needed. Communication between MemRed and browser pages is done using the ChromeDevTools protocol [2]. In the following, we describe the different components of MemRed.

### 3.1 Monitoring

In the monitoring phase, MemRed connects to a page and collects memory-usage data. As shown in Figure 1, a browser tab is a separate process having three components: the Javascript engine which is called V8 in Chrome and executes the Javascript code of a web application, the DOM [1] bindings which are responsible for binding Javascript engine and browser code, and the HTML renderer which displays the web page on the screen based on HTML code and cascading style sheets (CSS) of the web application. MemRed collects snapshots of the Javascript heap periodically, which consists of the heap graph structure as well as the size of objects. The heap snapshot also contains information about the DOM objects of the page. Table 1 shows metrics collected from the Javascript and DOM domains. We also monitor the amount of memory allocated to the heap by instrumenting the V8 source code. It is shown in Table 1 under category JS Memory. It is worth mentioning that the ChromeDevTools protocol performs full garbage collection before taking a heap snapshot.

In the Chrome browser different tabs may share the same process and in such cases the Javascript heap contains the objects belonging to several tabs. We need to differentiate between objects of different tabs, even though they share the same process and Javascript engine. Currently, the ChromeDevTools protocol does not provide the information regarding which object belongs to each tab. We need to augment this protocol to get tab related information by taking advantage of information entailed in V8 Javascript engine. In Firefox browser, the objects belonging to different applications have been separated using the compartment idea [26].

[1]The Document Object Model (DOM) is an abstract representation of the HTML page and provides an API for accessing and manipulating HTML elements.

| Monitored Component | Metric | Description | Aging Indicator |
|---|---|---|---|
| **Javascript Heap** | **TotalSize** | Total size of all objects on the Javascript heap. | ✓ |
| | NodeCount | Total number of all objects of the application, including objects on the Javascript heap as well as browser objects. | × |
| | ArraySelf | Sum of shallow sizes of all objects that are instantiated using the array constructor function. | × |
| | ArrayCount | Total number of objects that are instantiated using the array constructor. | × |
| | HiddenSelf | Sum of shallow sizes of hidden objects (objects that are created behind the scenes by V8 (Javascript engine of the Chrome [7]). | × |
| | HiddenCount | Number of hidden objects. | × |
| **DOM data** | DocDomCount | Number of DOM objects of the page. | × |
| | DetachDomCount | Number of objects that are detached from the DOM tree of the page. | × |
| **JS Memory** | TotalHeapSize (JS) | The amount of memory allocated to Javascript heap by V8. | × |

**Table 1: Metrics collected from different domains of a browser tab.**

## 3.2 Error Detection

In this phase, the collected data is analyzed to detect any sign of error. Specifically, we want to detect excessive memory usage. Therefore, we analyze the `TotalSize` metric, shown in Table 1, which indicates the memory usage of a web application. Other metrics are useful for the diagnosis step; however, in this prototype we do not have a diagnosis step and leave it for future work.

To find the trend in the specified metric, we use linear least squares method [27] to fit a line over data and measure the quality of this fit by computing the correlation coefficient [27]. If the memory usage of a web application is continuously going up, this is reflected in the slope of the line fitted to the data. A positive slope together with a high correlation coefficient shows an increasing trend in data. The higher is the correlation coefficient, the more accurate is the slope of the line fitted to data. We cannot make any conclusion about existence of any trend in the data if correlation coefficient is low. The slope and correlation coefficient are threshold values that should be determined based on the desired level of sensitivity.

Note that the above technique is able to detect errors if the data is collected from the same state of an application. We define the state from the user's point of view. Two application states are the same if they look the same to the user. We estimate an application state by the DOM tree of the application. Continuous growth of the collected metric over the same states is a sign of error. In this work, we are not monitoring application states and assume that we are aware of the state at each data collection point.

## 3.3 Recovery

In this phase we apply recovery actions to return the system to a healthy state or to slow down system degradation. The recovery actions should be applied at an appropriate time to hide the failures from the user. Moreover, the recovery action should be transparent to the user. The recovery actions selected depends on the cause of errors. As mentioned earlier, in the current prototype we do not have a diagnosis step and leave it for future work. In this work, we study the effectiveness of the following recovery actions which are implemented using the Chrome extension APIs:

- **Page Reload**: This action refreshes the web application, deleting all the leaked and unused objects.
- **Close the page and open it in a separate process**: This action restarts the process corresponding to the page.
- **Full Garbage Collection**: This action collects all the garbage on the Javascript heap of the application.

To reload or restart a page, MemRed waits until the monitored tab is not visible to the user; garbage collection can be done at any time.

## 4. EVALUATION

In this section, we examine the effectiveness of recovery actions in improving the reliability of web applications. To perform these experiments, we use two web applications. The first one, Tudu [6], is an AJAX-based open source web application which is used for managing personal todo lists. The client side of this application consists of 3K lines of code as well as AJAX libraries [19]. The second one is Google's GMail [3] application, which is developed mostly using Javascript and AJAX technology. Tudu is an example of a simple and small web application in which we can simply inject bugs. On the other hand, GMail is one of the largest web applications in wide-spread use that is developed by high-quality engineers.

In every experiment, we monitored a web application running in a browser tab while a user navigates through the page and generates events. To expedite the effects of software

| | Description |
|---|---|
| **TuduNoAction** | Simulated user is navigating through the Tudu page and doing the set of actions repeatedly. |
| **TuduReload** | Simulated user is navigating through the Tudu page similar to the TuduNoAction case; the pages is reloaded periodically. |
| **TuduDiff** | Simulated user is navigating through the Tudu page similar to the TuduNoAction case; the page is closed and opened in a new tab (actually in a new process) periodically. |
| **GmailNoAction** | Simulated user is navigating through the page and doing the set of actions repeatedly; no recovery action is applied. |
| **GmailReload** | Simulated user is navigating through the page like GmailNoAction case; the page is reloaded periodically. |
| **GmailDiff** | Simulated user is navigating through the page as the GmailNoAction case; the tab is closed and opened in a new tab (actually in a new process) periodically. |
| **GmailIdle** | Gmail page is open and being monitored but the simulated user does not do any action. |
| **GmailIdleFullGC** | Gmail page is open and being monitored; the simulated user does not do any action; periodically a full garbage collection is performed. |

**Table 2: Different experiments performed on Tudu and GMail applications.**

aging we simulated user navigation using the WebDriver API [4] for Java. To alleviate the need for application-state monitoring, as discussed in Section 3, we forced the simulated user to do a set of repeated tasks. Resource usage over time for such a scenario is good indication of software aging.

The experiments in this section are performed on a 2.4 GHz quad-core PC with $4GB$ of RAM. We use Weka [8], an open source machine-learning library, for data analysis. The different experiments are described in Table 2 and corresponding results are discussed in the following. First, we study the case where we injected a leak in the Tudu application. Then, we show the result of experiments on the GMail application.

## 4.1 Tudu Experiments

After logging into the Tudu page, the simulated user does the following actions iteratively while waiting 4 seconds between each action:

1. Add a new todo list.
2. Edit the list.
3. Delete the list.

At all times, there are at most two lists in the system. We injected a bug into Tudu application, so that every time the user adds a list, some objects are created and added to the Javascript heap, but are not used again. This results in a memory leak in this application which is an example of a software aging error. Note that since Javascript is a garbage-collected language, a memory leak means the existence of unused objects; i.e., objects that are accessible from the garbage-collection roots but are never used by the application [22].

Figure 2 shows the variation in total size of objects on the heap (`JSLive`) for different experiments on Tudu page. The period of data collection is one minute. Note that software aging in our system is a function of the number of events on the page. However, since the user is repeating the same set of tasks, time is a good approximation of the number of events on the page. As shown in Figure 2, in the first experiment, TuduNoAction, when no recovery action is applied; we see an increasing trend in size of objects on the heap, because of the leak.

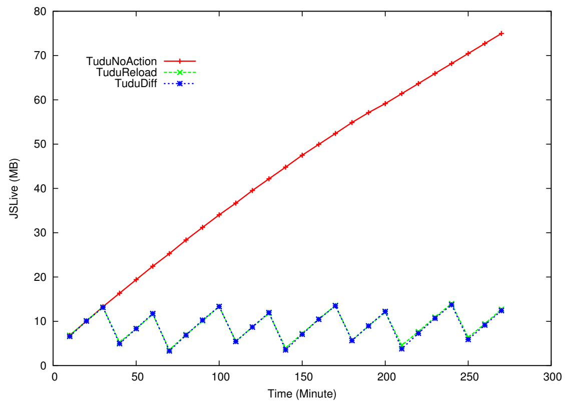

**Figure 2: Variation in total size of objects on the heap (JSLive) over time for different experiments on Tudu.**

To see the effects of recovery actions, we perform TuduReload experiment, in which we periodically reload the page, and TuduDiff, in which we periodically close and open the tab in a new process. As shown in Figure 2, there is not an increasing trend in these cases. The oscillation in data values is due to performing a recovery action periodically; i.e., every 37 minutes in this case.

We also fitted a line to the collected data to compute any trend in the data. The slopes and correlation coefficients of the corresponding lines are shown in Table 3. A positive slope shows increasing trend if it is accompanied by a correlation coefficient. To compute the slope for TuduReload and TuduDiff experiments, we smoothed the data with ten neighboring points to remove the effect of data oscillation on trend estimation. As can be seen, we do not see any trend in the data when recovery actions are applied periodically. These results support the idea that web application's relia-

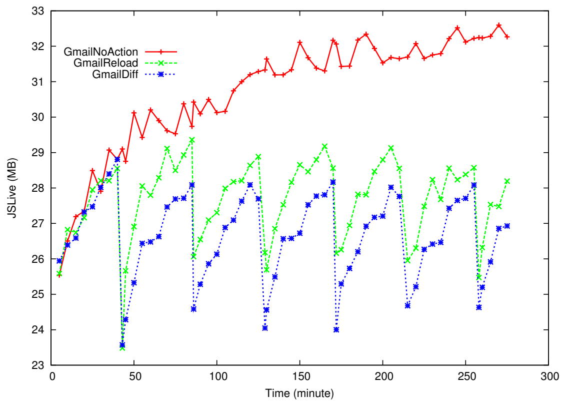

**Figure 3: Trend in total size of objects on the heap (JSLive) for different experiments on GMail.**

bility can be improved by our approach. Recovery actions prevent excessive memory usage by a web application, so more memory is available to other tabs. This means better response time and time-to-failure for the whole system.

## 4.2 GMail Experiments

The simulated user, after logging into her GMail account, performs the following actions iteratively while pausing a few seconds between every action:

1. Compose an email (just clicks on the compose without sending any email).
2. Click on the list of important emails.
3. Click on Inbox.
4. Read an email (the same email is read in each iteration).

As shown in Figure 3, where no recovery action is applied on the page, i.e. in the GmailNoAction experiment, the total size of objects on the heap is going up. In this experiment, the user is doing a repeated set of tasks without generating new objects (she is not sending or receiving any email). Therefore, growth in the size of objects indicates an aging issue in the GMail application. Since the ChromeDevTools protocol performs a full garbage collection before taking a heap snapshot, this growth cannot be related to a delay in garbage collection. This behaviour may be due to a memory leak; the important point is that this application suffers from excessive memory usage. It is possible that the application is intended to function in this way. For example, the application may keep a history of all the states; the user may not refer to them in the future, which results in a growth in the number and size of the objects on the heap. This behaviour is not acceptable in an operational context since it results in unbounded memory growth which affects the reliability and response time of the system. Therefore, having a mechanism to reduce excessive memory usage is crucial.

Figure 3 also shows the result of the GmailReload experiment, where we periodically reload the page, and the GmailDiff experiments, where we periodically close and open the page.

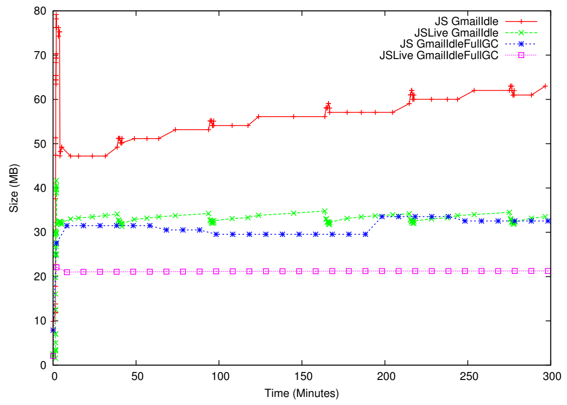

**Figure 4: Trends in total size of objects on the heap (JSLive) and memory allocated by Javascript engine for the page (JS) for GmailIdle and GmailIdle-FullGC experiments.**

| | JSLive | |
|---|---|---|
| | **Slope** | **CC** |
| **TuduNoAction** | 0.26 | 0.9976 |
| **TuduReload** | 0 | 0.1283 |
| **TuduDiff** | 0 | 0.1267 |
| **GmailNoAction** | 0.017 | 0.7011 |
| **GmailReload** | 0 | 0.306 |
| **GmailDiff** | 0 | 0.0807 |
| **GmailIdle** | 0 | 0.1019 |

**Table 3: Slope and correlation coefficient (CC) of JSLive for different experiments, unit of slope is MB/minute.**

We do not see an increasing trend in these cases, since the recovery actions delete objects from the heap. Table 3 shows the slopes and correlation coefficients of the lines fitted to the data. To compute the parameters of the line fitted to data we smoothed each data point using ten neighboring points. The reason is that the data values for GmailReload and GmailDiff are oscillating between two boundaries because of periodic recovery actions. As can be seen, there is no increasing trend when the recovery actions are applied.

To measure the effectiveness of the garbage collection action we need to look at the memory allocated by V8, which is measured by the TotalHeapSize (JS) metric, as defined in Table 1. To study this action, we perform two experiments where the user is idle in both cases. In GmailIdleFullGC we periodically collect all garbage and in GmailIdle we apply no recovery action. Figure 4 shows the values of `JSLive` and `JS` metrics over time. As can be seen, the memory that is allocated by V8 (JS) is going up for the GmailIdle experiment; however, we do not see such a trend when garbage collecting periodically. Also, there is no increasing trend in the size of objects on the heap (JSLive). Note that even though the simulated user is idle, there are some events fired on the page using a timer.

The increasing trends observed in metrics collected from the GMail application represent software aging in a commonly used web application where the user is doing a set of simple tasks. In a real world scenario in a browser, we are dealing with large number of tabs, each one executing a different web application. We also need to consider that GMail is a well-engineered application and that is not the case for all those applications running in different tabs. Therefore, we need a mechanism for improving reliability of the client side of web applications.

## 5. CONCLUSION AND FUTURE WORK

In this paper, we have presented an initial implementation of our approach for improving the reliability of the client side of web applications by reducing memory usage. MemRed, which is an extension in the Chrome browser, monitors the Javascript heap of a web page. It then analyzes the collected data to detect trends in memory usage of the corresponding web application. Upon detecting a possible memory leak, it applies a recovery action at an appropriate time. Through experiments, we showed the effectiveness of our recovery actions in lowering the memory usage of web applications. However, there are many limitations with the current prototype that need to be addressed in the future. We need to enrich our recovery actions so that the user does not lose any data because of recovery actions. This can be done by monitoring the application state. In addition, a diagnosis step is needed to take advantage of all the data collected from different domains of a page.

## 6. REFERENCES

[1] Chrome browser. *https://www.google.com/chrome.* Visited: May 2012.

[2] ChromeDevTools protocol. *http://code.google.com/p/chromedevtools/wiki/ChromeDevToolsProtocol.* Visited: May 2012.

[3] Gmail: A Google approach to email. *http://gmail.com.* Visited: May 2012.

[4] Selenium 2.0 and webdriver. *http://seleniumhq.org/docs/03_webdriver.html.* Visited: May 2012.

[5] StatCounter Global Stats. *http://gs.statcounter.com/.* Visited: May 2012.

[6] Tudu lists. *http://tudu.sourceforge.net/.* Visited: May 2012.

[7] V8 JavaScript engine. *http://code.google.com/apis/v8/design.html.* Visited: May 2012.

[8] Weka–data mining with open source machine learning software. *http://www.cs.waikato.ac.nz/ml/weka/.* Visited: May 2012.

[9] J. Alonso, J. Torres, J. Berral, and R. Gavalda. Adaptive on-line software aging prediction based on machine learning. In *Proceedings of the 40th International Conference on Dependable Systems and Networks*, 2010.

[10] S. Andrica and G. Candea. WaRR: A tool for high-fidelity web application record and replay. In *41st International Conference on Dependable Systems and Networks*, 2011.

[11] G. Candea, S. Kawamoto, Y. Fujiki, G. Friedman, and A. Fox. Microreboot–a technique for cheap recovery. In *Proceedings of the 6th Symposium on Opearting Systems Design & Implementation*, 2004.

[12] B. Cully, G. Lefebvre, D. Meyer, M. Feeley, N. Hutchinson, and A. Warfield. Remus: High availability via asynchronous virtual machine replication. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, 2008.

[13] S. Garg, A. Van Moorsel, K. Vaidyanathan, and K. Trivedi. A methodology for detection and estimation of software aging. In *Proceedings of the 9th International Symposium on Software Reliability Engineering*, 1998.

[14] M. Grottke, L. Li, K. Vaidyanathan, and K. Trivedi. Analysis of software aging in a web server. *IEEE Transactions on Reliability*, 55(3), 2006.

[15] S. Guarnieri and B. Livshits. Gatekeeper: Mostly static enforcement of security and reliability policies for Javascript code. In *Proceedings of the 18th Conference on USENIX Security Symposium*, 2009.

[16] Y. Huang, C. Kintala, N. Kolettis, and N. Fulton. Software rejuvenation: Analysis, module and applications. In *Proceedings of the 25th International Symposium on Fault-Tolerant Computing*, 1995.

[17] M. Isard. Autopilot: automatic data center management. *ACM SIGOPS Operating Systems Review*, 41(2), 2007.

[18] E. Kiciman and B. Livshits. Ajaxscope: a platform for remotely monitoring the client-side behaviour of web 2.0 applications. *ACM SIGOPS Operating Systems Review*, 41(6), 2007.

[19] A. Mesbah. *Analysis and Testing of Ajax-based single-page web applications*. PhD thesis, Delft University of Technology: TU Delft, 2009.

[20] J. Mickens, J. Elson, and J. Howell. Mugshot: Deterministic Capture and Replay for Javascript Applications. In *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation*, 2010.

[21] F. Ocariza Jr, K. Pattabiraman, and B. Zorn. Javascript errors in the wild: An empirical study. In *Proceedings of 22nd International Symposium on Software Reliability Engineering*, 2011.

[22] D. Rayside and L. Mendel. Object ownership profiling: a technique for finding and fixing memory leaks. In *Proceedings of the 22nd International Conference on Automated Software Engineering*, 2007.

[23] C. Reis and S. Gribble. Isolating web programs in modern browser architectures. In *Proceedings of the 4th European conference on Computer systems*, 2009.

[24] G. Richards, S. Lebresne, B. Burg, and J. Vitek. An analysis of the dynamic behaviour of javascript programs. *ACM SIGPLAN Notices*, 45(6), 2010.

[25] S. Sidiroglou, O. Laadan, C. Perez, N. Viennot, J. Nieh, and A. Keromytis. Assure: automatic software self-healing using rescue points. *ACM SIGPLAN Notices*, 44(3), 2009.

[26] G. Wagner, A. Gal, C. Wimmer, B. Eich, and M. Franz. Compartmental memory management in a modern web browser. *ACM SIGPLAN Notices*, 46(11), 2011.

[27] R. Walpole, R. Myers, S. Myers, and K. Ye. *Probability and statistics for engineers and scientists*, volume 5. Macmillan New York, 1972.