JS Capsules: A Framework for Capturing Fine-grained JavaScript Memory Measurements for the Mobile Web

USAMA NASEER, Brown University, USA
THEOPHILUS A. BENSON, Brown University, USA and Carnegie Mellon University, USA

Understanding the resource consumption of the mobile web is an important topic that has garnered much attention in recent years. However, existing works mostly focus on the networking or computational aspects of the mobile web and largely ignore memory, which is an important aspect given the mobile web's reliance on resource-heavy JavaScript.

In this paper, we propose a framework, called JS Capsules, for characterizing the memory of JavaScript functions and, using this framework, we investigate the key browser mechanics that contribute to the memory overhead. Leveraging our framework on a testbed of Android mobile phones, we conduct measurements of the Alexa top 1K websites. While most existing frameworks focus on V8 — the JavaScript engine used in most popular browsers — in the context of memory, our measurements show that the memory implications of JavaScript extends far beyond V8 due to the cascading effects that certain JavaScript calls have on the browser's rendering mechanics. We quantify and highlight the direct impact that website DOM have on JavaScript memory overhead and present, to our knowledge, the first root-cause analysis to dissect and characterize their impact on JavaScript memory overheads.

CCS Concepts: • **Networks** → **Network performance analysis**; **Network measurement**; **Mobile networks**.

Additional Key Words and Phrases: Mobile web; Memory performance; Web optimizations.

ACM Reference Format:
Usama Naseer and Theophilus A. Benson. 2023. JS Capsules: A Framework for Capturing Fine-grained JavaScript Memory Measurements for the Mobile Web.. *Proc. ACM Meas. Anal. Comput. Syst.* 7, 1, Article 14 (March 2023), 27 pages. https://doi.org/10.1145/3579327

1 INTRODUCTION

Mobile phones have become the primary gateway to the web, with the number of mobile subscriptions surpassing the total human population by 1.3 Billion [103]. Despite bringing equitable Internet access to the developing regions, the mobile web is still plagued with performance issues, e.g., the page loads time on mobile devices are still orders of magnitude slower than on desktops [18, 81]. Studies suggest that Quality-of-Experience (QoE) problems on the mobile web arise due to the friction between the high resource demands of complex web sites [17] and the modest hardware of mobile phones [2, 23, 101]. This plight is even more pronounced for users in developing regions where low-end phones still hold a major market share [2, 6] and their connectivity lags far behind the developed world [12, 56]. Consequently, and due to the tight coupling of QoE with revenue [15, 16, 46, 59, 86], improving mobile-web performance has recently garnered a lot of attention.

Authors' addresses: Usama Naseer, usama_naseer@brown.edu, Brown University, Providence, RI, USA; Theophilus A. Benson, theophilus@cmu.edu, Brown University, Providence, RI, USA and Carnegie Mellon University, Pittsburg, PA, USA.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.
2476-1249/2023/3-ART14 $15.00
https://doi.org/10.1145/3579327

Recent studies have identified a number of factors that lead to poor mobile web performance, ranging from network [5, 69, 83], computational limitations [33, 81, 101], protocol limitations [51, 80, 84, 111], and energy wastage [19] to JavaScript inefficiencies [21–23, 69, 72]. While there is significant work on understanding the mobile device's limitations, surprisingly, one key resource is missing from these works: specifically, little is known about the memory overhead of client-side JavaScript. Anecdotal evidence suggests that memory constraints in the mobile devices and the increased complexity of client-side JavaScript [8] have a direct impact on both the performance (e.g., due additional computational overhead of garbage collection) and availability of web pages (e.g., browser crashes). Understanding the memory overhead of JavaScript is especially important, since JavaScript is a key part of today's web ecosystem and plays a critical role in building interactive websites [21–23, 69, 88].

Investigating JavaScript memory overhead requires holistic measurement techniques to capture fine-grained allocations (i.e., *how much memory is allocated?*) and identify their source (i.e, *what and why the JS functions contributed memory?*). However, the nature of JavaScript's dynamic interactions with a webpage's Document Object Model (DOM) presents a significant challenge towards gaining a holistic view of memory because these dynamic interactions often cascade into events that span multiple browser components. For instance, dynamically fetching and adding an image to the DOM requires memory for networking components to fetch and store the image, computations (e.g., resizing) on V8 (i.e., the JavaScript engine used in most browsers) and rendering components for computing the updated layout and printing the pixels on screen. Thus, JavaScript incurs a *direct* and an *indirect* memory overhead, and a fine-grained, cross-component view is required to correctly attribute the memory overhead to the respective JavaScript functions.

Unfortunately, existing OS-level [38, 90] and Chrome-based [10, 43, 67] memory analysis tools either provide coarse-grained visibility or incomplete coverage. Consequently, they are unable to dissect the memory implications of JavaScript functions and do not provide enough fine-grained information to tie memory allocations to their root-cause (e.g., local objects, API calls, render events, etc.). While some JavaScript memory profilers, e.g., *HeapProfiler* [10], provide fine-grained visibility into the V8 engine to highlight the potential root-causes, their scope is only limited to the V8-component and, thus, they do not provide complete coverage over JavaScript's indirect memory overhead. In fact, our measurements indicate that, for the median website, such profilers underestimate JavaScript memory by a factor 3.7X. Consequently, existing tools lack the ability to localize the potential reasons for JavaScript direct and indirect memory allocations.

In this paper, we propose a novel measurement framewok for efficiently and accurately capturing fine grained measurements of JavaScript functions' memory overhead. The key insight behind our design is that most browsers execute JS in a single-threaded fashion [74, 82, 101] per frame and use a single *main* thread for scheduling JS events [74] to guarantee safe concurrency for the shared DOM data structure. Leveraging this property, we decompose a website's JS into isolated blocks called *JS Capsules*, formally defined as a time interval (i.e., *start* and *end* timestamps) that encapsulates one or more JS functions. Leveraging the timestamps and the serial-access model, we capture a JS Capsule's cascading events by decomposing the page load process along a browser's thread boundaries, and use low-level Chromium APIs [40, 41] to measure fine-grained memory allocations. In essence, JS Capsules provide us with a framework for fine-grained visibility into the JavaScript functions and the resulting cascading events, as well as, a complete coverage over the various sources of memory allocations. We make the following contributions:

- We present the first-ever characterization of JavaScript memory for the Alexa top 1K websites. Our measurements indicate that JavaScript can incur as much as a 250MBs overhead for certain

categories such as *news* and *sports*. Surprisingly, the indirect consequences of JavaScript contribute the major share (73% for the median website).
- Analyzing the key culprits of memory bloat, we identify JavaScript's dynamic interactions with the DOM and the cascading rendering-related events as a key source. In fact, the dynamic interactions incurs a 6X memory overhead, as compared to the static DOM interactions through HTML.
- We characterize the memory overhead of advertisements, trackers and general-purpose libraries in popular websites. Ads and trackers cumulatively contribute over 20% of JavaScript memory for the median *news* and *sports* websites, while libraries contribute a higher share (up to 24%). We dissect such JavaScript functions and their impact on DOM to explain their memory overhead trends.
- To identify the key browser mechanics behind the JavaScript memory overhead, we classify the JavaScript functions based on their cascading events and highlight several unique characteristics across different websites. Leveraging this framework, we visualize the key differences between the websites with low or high JavaScript memory usage.

The paper is structured as follows: Section 2 presents background on Android OS memory management, Chromium architecture and the page load process and Section 3 presents the motivation for JS memory analysis. Section 4 discusses the challenges associated with capturing fine-grained memory analysis using existing tools and presents our framework (JS Capsules). Section 5 discusses the implementation details of JS Capsules, our testbed and the various analysis techniques we leverage for the memory attribution, while Section 6 validates our methodology. In Section 7 and Section 8, we present the measurement results and the root-cause analysis to identify the key JavaScript features that impact memory. Finally, in Section 9, we discuss the impact of our work on several web optimizations.

2 BACKGROUND

In this section, we discuss Android OS's memory management, the architectural details of the Chromium browser and provide a high-level overview of the page load process, with a specific focus on JavaScript. We also discuss the role of different Chromium memory allocators for loading and rendering a webpage.

2.1 Android Memory Management

Android runs on the premise that free memory is "wasted" memory and tries to use all of the available memory [34] at all times [36]. Unlike traditional Linux, Android does not use virtual memory techniques (e.g., swapping) to ease memory contention and instead relies on Garbage Collection (GC) [37] and Out-of-Memory (OOM) killer [68]. Garbage collection is a mechanism for reclaiming unused memory and, at a high-level, has two goals: (i) find memory references that cannot be accessed in future, and (ii) reclaim such memory objects [37]. Garbage collection often incurs CPU overhead and may even stall the application [50]. In the extreme cases when enough memory cannot be reclaimed, Android uses OOM killer to identify and terminate one or more processes (based on pre-configured priorities), thereby reclaiming memory space for a higher-priority process.

2.2 Chromium Architecture

Chromium is the underlying framework that powers popular browsers like Chrome, Brave, Edge, Opera, etc. [25]. In 2022, Chromium accounted for over 67.6% of the global mobile browser share [97]. Given Chromium's broad use, we believe that by building our design on its architecture, we ensure

that our insights and observations are broadly applicable across popular browsers. On Android, Chromium uses a multi-process architecture with three types of processes — *Browser*, *GPU* and *Renderer* — which are responsible for different aspects of loading a webpage [24, 70].

- The Browser process is in charge of UI, disk, and network I/O and, in particular, controls the address bar, browser UI buttons, file access and makes network requests.
- The GPU process interfaces with the GPU device for GPU tasks. The architecture also specifies an in-process GPU implementation [98] and selectively offloads tasks to the device GPU.
- The Renderer process "controls anything inside of the tab where a website is displayed" [70] and performs the core functionality of parsing and rendering webpages. It includes JS compilation engine V8 [52, 100]. It also parses webpage code (e.g., HTML, CSS, JavaScript) and prints the webpage on screen. While the other two processes are shared across different tabs, a new Renderer is spawned for each new tab. Furthermore, for security and isolation purposes, the browser spawns multiple Renderers if the webpage contains cross-origin content (typically in iframes) [71, 75].

2.3 Page Load Workflow

The page load process begins with a user-initiated request which downloads the root HTML file. The HTML file specifies tags for different content types (e.g, *a or p* for text, *img* for images, *script* for JavaScript) and a hierarchy, i.e., sibling or parent-child relationships between the different tags. The HTML file may also include references to other web objects (e.g., URLs for images, JS/CSS files).

As soon as the first chunk of HTML file is downloaded, the Renderer initiates *HTML parsing* to construct the *Document Object Model* (DOM) [29], an object-oriented representation of a webpage's structure and content. The DOM comprises of nodes for the different tags and is constructed as a tree to maintain the hierarchy between the nodes. The Renderer evaluates the styles for the DOM nodes (typically defined through style attributes in HTML or CSS) to yield a *render tree*, which serves as the basis for determining the layout of nodes (i.e., position and size) and painting the visible nodes on screen. The Renderer progressively paints the webpage while parsing the HTML. While parsing, the Renderer also downloads the referenced objects (via the Browser process). However, these objects are only evaluated when needed [101].

Among the referenced objects is JavaScript (JS for short), the key focus of this paper. At its core, JS provides a programmatic interface for dynamically interacting with the DOM. Developers use JS to dynamically fetch objects, add new nodes, modify existing nodes (e.g., position, size), modify style rules (e.g., CSSOM [28]) or perform any required computations (e.g., calculations, maintaining sessions or state). To avoid any conflicts in building or modifying the DOM tree [101], the browser blocks HTML parsing while any embedded JS is evaluated. The DOM updates by JS API calls (e.g., modify an existing node) can have a cascading effect leading to the Renderer re-generating the render tree, recalculating the layout, and appropriately generating new frames to paint on the screen [73].

JS is a single-threaded language [54]. Moreover, Chrome instantiates a separate instance of the V8 engine within each Renderer which, in turn, maintains a single memory heap (used to allocate memory for the JS program) and call-stack [104]. Browsers executes JS in a single-threaded fashion [74, 82, 101] per frame and use a single *main* thread for scheduling JS and most rendering events [74]. This *serial-access* model simplifies page development and guarantees concurrency for the shared DOM data structure.

Actions	Mapped Allocators
Network requests	Memory allocated to connections (net), Downloaded media (web_cache)
HTML parsing	General-pupose allocators for storing trees (malloc, partition_alloc)
JS evaluation	Memory used by V8 engine (v8), General-pupose allocators for storing data-structures (malloc)
Rendering/Painting	General-pupose allocators (malloc, partition_alloc, shared_memory), Compositor (cc), GC for rendering engine (blink_gc), Memory used for GPU purposes (gpu), Skia rendering system (skia)
Misc	Fonts (font_cache), Local databases (level_db, sqlite, shared_memory), Browser UI (ui), Account sync (sync)

Table 1. Memory allocators used by the different browser actions performed during the page load process.

2.4 Chromium Memory Allocators

While loading the page, Chromium uses a number of different allocators for allocating memory to any of its processes. Table 1 lists the different actions performed by the browser during the page load process and the respective allocators responsible for memory allocations. While some of the allocators have distinct one-to-one mapping with page load actions, e.g., *net* and *V8* are always used for network requests and JS evaluation respectively, others such as *malloc, partition_alloc* and *shared_memory* are used across the different page load actions. Furthermore, the memory allocated as a result of executing a JS function may span multiple allocators. As JS DOM API calls [76] require access to the DOM, Chromium stores the references to DOM nodes in the JS heap (V8 heap [66]), while the actual objects and metadata of DOM nodes are stored within the Renderer process [10, 65, 66], e.g., reference to an image is stored in DOM while the actual image and its bitmap representation are stored in the Renderer.

3 MOTIVATION

The mobile web is an important part of the web ecosystem, contributing over 55.8% of website traffic worldwide in 2022 [26]. While mobile apps, such as apps from Facebook, CNN, Amazon etc., are a key driver behind the high mobile traffic share, mobile browsers also play a key role in this ecosystem. A 2017 study of 300 top retailers showed that 51% of retailers do not have a mobile app and, instead, rely completely on mobile websites for their sales [63]. Furthermore, reports suggest that Internet users in developing regions use mobile phones as the primary gateway to the web [11]. As web performance highly impacts revenue [15, 16, 46, 86], the high share of mobile web traffic shows that there is a strong incentive for content providers to improve performance.

In this section, we discuss the implications of memory on web performance (Section 3.1) and present a longitudinal measurement to show that JS memory bloat has increased over the years, thus motivating techniques that can help in understanding memory's overhead (Section 3.2).

3.1 Performance Implications of Memory

Performance impact of memory is device-specific and highly depends on the amount of available memory [34]. A recent work reported that when the device memory is reduced from 2GB to 512MB [33] then page load time inflates by a factor of 2X. Such contention is expected to be more frequent for low-end devices, e.g., up to 48% of the available memory is allocated to the native OS for a 0.5GB device (measured for Android 7.1). We observe memory-heavy websites, such as *news* and *sports*, to be problematic as they (together with the OS) can force the device to use over 90% of its memory.

There are primarily two implications of memory contention on the web performance. First, garbage collection incurs a compute overhead and reduces CPU cycles available for loading the webpage, and may momentarily pause browser application to reclaim space. We measure that at the median, garbage collection paused the browser for a 2.6X longer duration (472ms vs. 181ms) on a 0.5GB vs 1GB device for the Alexa top 50 US pages (measured through *logcat* [45]). Total reported garbage collection time (pause plus background garbage collection) differed by a staggering 4.8s at the median. Second, in the worst case, page loads can overwhelm the available memory, and

such contention can crash the browser or tab itself via the OOM killer. Studies suggest that users often have multiple tabs opened within their browser for productivity reasons and are constantly switching between tabs [20]. We emulated such a scenario and observed that loading a single news page (e.g., elwatannews.com, elbalad.news, youm7.com, dailymail.co.uk) led to crashing all of the background tabs, even for a device with 1 GB of RAM.

3.2 Longitudinal Study of Memory Usage

While mobile adoption has steadily increased over the years, studies suggest that mobile hardware resources vary widely across different regions of the world [3, 4, 6, 7, 14, 87, 93]. Due to economic and availability factors, low-end devices account for a higher share in developing regions, with a 2020 study showing that low-end devices (1GB or less RAM) have 3× more market share in developing regions (57% are low-end, 25% have 0.5GB or less RAM) as compared to developed regions (20% are low-end, >1% have 0.5GB or less RAM) [2]. One would expect that, ideally, this stark difference in the memory landscape should be reflected in the memory expectations of different regional websites.

To investigate this conjecture, we analyzed the memory utilization for 250 popular websites over a period of 6 years, from 2015 to late 2021. Our list of 250 websites comprise Alexa top 50 websites (list collected in August 2020) from 5 countries: Egypt, Nigeria, Pakistan, U.S. and U.K. (250 websites in total). These countries represent five of the most densely populated regions with the highest number of web users, and provide a good mix of regionally and globally popular websites (31% of websites are exclusive to EG, NG, PK). We leveraged the *WayBack Machine* from the *Internet Archive* [9], and measured page load memory[1] for the historical snapshots of the respective websites. For each website, we captured the recorded snapshots from each quarter of the year, randomly select one snapshot per-quarter, and repeat the measurement ten times. This process is repeated for every year, from 2015 to 2021, and the webpages are loaded on an android mobile phone (testbed discussed in Section 5.2). To validate that the webpage snapshots from *WayBack Machine* are representative, we measured the memory of actual respective websites (i.e, live websites in 2021) and compare live memory with the 2021 snapshots. We observe that the memory for snapshots is within 4% margin of error for the median website (9% at the tail)[2].

Figure 1 plots the average memory per-year for the developing, developed and all regions[3], normalized by the average memory in 2015 across all regions. We observe that the memory footprint has steadily increased over years globally. Interestingly, the average memory for developing region websites tends to be higher than developed region ones (though the difference is smaller than ever in 2021) and this trend stands in stark contrast to the device memory landscape across regions. More importantly, we observed JavaScript to be the key culprit behind memory. In Figure 2, we divide the websites from 2021 into four quartiles and plot the percentage of memory single-handedly contributed by JS (calculating ground-truth JS memory in Section 6). For the websites with highest memory (i.e., Q4), JS single-handedly contributes 58.5% of the memory at the median (over 77% at the tail). While one may safely assume that the steady increase over the years can be attributed to the increased use of client-side JS [8], the existing literature and tools fail in precisely answering

[1]We define "page load memory" as the max memory consumed by a browser tab as measured using *meminfo* [57] tool.
[2]Though we validate that the 2021 WayBack snapshots are representative, our experiment assumes that the prior webpage snapshots are correct. There are two caveats that can impact the snapshot correctness: (i) a web object (e.g., JS file) referenced in a historical snapshot may not be valid anymore, and (ii) loading same website from different regions may fetch different responses, e.g., *google.com* loads the regional page. For the latter caveat, although we use regional URLs (*google.com.pk* for Pakistan), we assume that the recorded snapshot is the correct regional version.
[3]Websites only exclusive to developing regions are classified as developing region websites, while websites popular in both regions are classified into the developed region class.

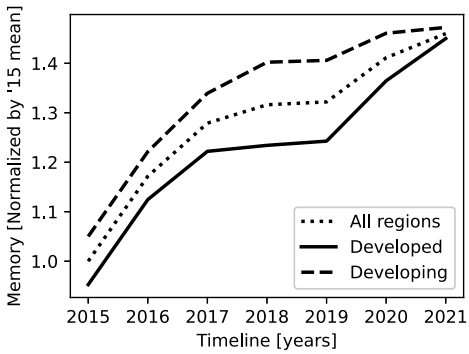


Fig. 1. Evolution of web memory, normalized by 2015.

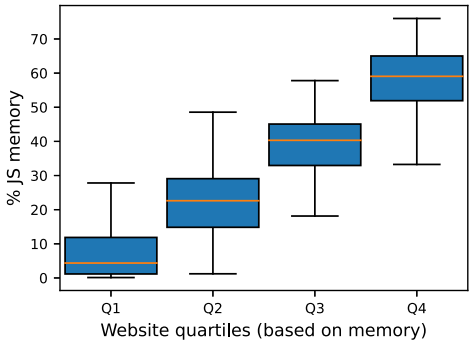


Fig. 2. Percent of JS contribution to total website memory.

the key enigma that why does this memory bloat exists today? Does the bloat exist due to certain JS calls or libraries? ads or trackers? specific website design choices? or, is there something more fundamental behind the memory bloat?

Takeaway: Memory's impact on performance and the significant memory contribution by JS motivate the need for automated tools to analyze and understand any potential inefficiencies. This knowledge is pertinent for developing services that may suit each device's limitations. In fact, companies like Google, Facebook and Uber have taken practical steps to address the resource deficiencies of low-end devices by developing specialized applications and OSes (e.g., FB-lite [53], Uber-lite [61], AndroidGo [7]). A more effective and efficient measurement framework or profiler can help improve the web in a more democratized fashion by enabling individual developers to independently address these limitations.

4 MEASUREMENT METHODOLOGY

In this section, we discuss the design goals of measuring and attributing the root-cause of JS memory, and highlight the limitations of existing tools (Section 4.1). Then, we present our approach for measuring a website's JS memory in a fine-grained manner and identifying the key culprits (Section 4.2).

```
function bar(input_text, device_props) {
  const boxes = document.querySelectorAll(".box")
  for (const el of boxes) {
    $(el).text(input_text);
    const width = get_width(device_props);
    $(el).width(width);
  }
}
```

Listing 1. JS cascading impact illustration.

Design Goals: To understand the requirements, let us consider the JS code snippet in Listing 1 where a function *bar()* uses jQuery to iterate over *boxes* in HTML and modify their text and style. A number of different allocators are involved for allocating memory during *bar()'s* execution. *V8* allocates memory for parsing and evaluating *bar()*, for calling jQuery APIs and for the runtime objects, such as the array *boxes*. When *bar()* modifies the *text* and *width* through jQuery calls, this

results in modifying the DOM (e.g., styles) and memory is allocated by *malloc* and compositor (*cc* allocator) to calculate the layout and generate the tiles on screen. Depending on the node dependencies in the DOM tree, the node addition may allocate further memory while recalculating the styles and layouts for the dependent nodes. Further, if *bar()* had involved networking transactions, e.g., fetching an image, allocators *net* and *web_cache* would allocate memory for the network transaction and storing the media content. Since executing *bar()* leads to these various memory allocations, we define the memory footprint or size of the function *bar()* as the sum of all memory allocations made in response to executing the respective function.

The ideal technique for measuring and attributing JS memory should meet the following goals:

- Executing a JS function involves a direct and an indirect consequence on memory, e.g., *bar()* (Listing 1) directly allocates memory in V8 to run the code, while indirectly allocates memory for rendering and painting the modified nodes. Thus, a cross-allocator view is required to ensure complete coverage.
- Executing a JS function (or API) may result in activity that span multiple components, e.g., Browser process for network activity, V8 and compositor components in the Renderer process for the DOM API calls and layout changes. Thus, accurately attributing the cause of memory allocations requires capturing cascading events (browser events triggered by a JS function), hence a holistic cross-component view is required.
- A website may comprise thousands of JS functions. Thus, accurate root-cause analysis requires fine-grained and precise memory measurements for individual functions.

4.1 Limitations with Existing Tools

There are two broad classes of tools for measuring memory: OS-level [38, 57, 90] and Browser-provided [10, 43, 67] tool. These tools either provide coarse-grained information, or do not have complete coverage over the various memory allocators.

OS-level tools: The OS-level tools (e.g., *meminfo* [57], *dumpsys* [38] and *perfetto* [90]) capture information at the *process level* and thus are unable to distinguish between the different functions (or objects) being processed or attribute memory allocations to specific web objects, browser activity or JS functions. Due to the lack of visibility into a browser process' inner mechanics (i.e., *why* is the memory allocated?), they provide coarse-grained information. One option is to augment the OS-level view with application instrumentation (e.g., timestamps of of a JS function call). However, synchronizing timestamps between OS and application readings can be challenging, e.g., *proc meminfo* reading often drift. Another option is to modify the webpage (e.g., disable a JS function, remove web objects etc.) and attribute the memory differences (with and without the function or object) as the function or object's memory. However, this approach has several drawbacks: (i) it is highly inefficient and time consuming because a website may have 100s of JS functions and web objects, (ii) it may lead to incorrectness because it make break or eliminate crucial dependencies between different functions in a webpage. We evaluated this strategy at a JS file level and found it untenable because the variance and error bars were much higher than the differences.

Chrome-level tools: Chrome provides three relevant tools: (i) *MemoryInfra* captures all the different allocators and heaps, (ii) *Timeline Profile* captures a time-series of JS heap and browser activity (e.g, network, rendering, and scripting events), and (iii) *HeapProfiler* captures JS heap broken down by JS functions and objects (e.g., strings or arrays). While *MemoryInfra* provides complete coverage over the multiple allocators it only captures a snapshot of memory at a timestamp and lacks information about how this state was reached due to its inability to associate website code and DOM events with memory allocations. In particular, it does not provide visibility into the JS calls or browser events which trigger them, and only provides an after-the-fact memory snapshot,

thereby, it lacks the necessary information required to understand the change in memory state. On the other hand, *Timeline Profile* and *HeapProfiler* only capture information relevant to the JS heap. Unfortunately, the JS heap is only a small fraction of the memory allocated during the page load process. In particular, media (images, video, etc.), render/layout/style information is not stored in JS heap and the cascading effects of JS lead to allocations in other parts of browser (e.g., Renderer and GPU process). To illustrate, in Figure 8, we plot the *V8* memory (recall that JS heap is part of V8) as the percentage of the total scripting memory footprint and observe it be orders of magnitude smaller than the total memory allocated (3X on average). From Figure 8, we observe that *V8* memory only contributes ~27% of the total JS memory for the median website.

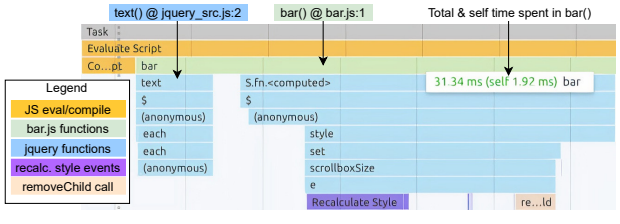
Thus, focusing solely on JS heap can greatly underestimate the total memory consumption of a website, as it omits the memory overhead of storing DOM, DOM content (e.g., images, text, etc.), and rendering. Our conversation with the Chrome developer indicated that although the report generated by *HeapProfiler* includes DOM node references, the tool itself does not measure the memory allocated to objects being referenced and, thus, only captures a subset of the overall memory footprint.

Takeaway: Existing tools are either too coarse-grained or do not capture sufficient information. Essentially using them either leads to over-estimation or under-estimation of JavaScript memory footprint.

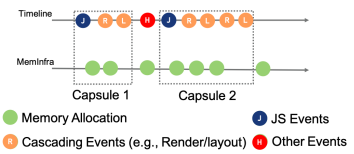
4.2 Our Framework - JS Capsules

At a high-level, our methodology is centered around decomposing a website's JS into isolated blocks, or *JS Capsules*, and capturing fine-grained memory, JS APIs and browser events for each JS Capsule. Formally, a JS Capsule is defined as a time interval (i.e., *start* and *end* timestamps) that encapsulates one or more JS functions. Leveraging a continuous memory timeline, we define the memory of a JS Capsule as the difference of respective memory snapshots at the JS Capsule's interval boundaries. Each snapshot includes the overall process memory, as well as, the allocator-wise breakdown.

The key insight behind our design is that browsers execute JS in a single-threaded fashion [74, 82, 101] per frame and use a single, *main*, thread for scheduling JS and most rendering events [74]. This *serial-access* model simplifies page development and provides safe concurrency guarantee for the shared DOM data structure. More explicitly, the model guarantees that a renderer process never executes more than one JS function in parallel and gurantees a sequential order for the rendering events, stimulated by a JS function. We leverage this guarantee to generate JS Capsules, compute their isolated memory footprint and attribute each browser event to the individual JS Capsules.



(a) Browser mechanics for executing *bar()* JS



(b) JS Capsules illustration

Fig. 3. JS Capsules are created by identifying JavaScript events and appropriate cascading events, then using their timestamps to capture the associated memory allocations.

4.3 JS Capsule Illustration and Design

To illustrate, Figure 3a presents the browser mechanics for executing the function *bar()* from Listing 1. The figure shows the call-stack for functions called within *bar()*, their timing information and browser events (e.g., *recalculate style*). Two time statistics are reported for each function, f(). The *self* time represents the time spent in f() and the *total* time represents the time spent in f() plus its dependents functions. In Figure 3a, we annotate the self and total time for *bar()*. The total time for *bar()* includes the time used by the jQuery functions used by *bar()*.

From a design perspective, there are two extremes for defining a JS Capsule: either use bar()'s total-time for marking the JS Capsule boundaries and aggregate the dependent functions and events, or use each f()'s self-time[4] to generate one JS Capsule per f(). While the second approach is more fine-grained, it requires capturing memory at a very high frequency. Yet, JS functions are short-lived, e.g., self-time for *style()* and *set()* is 3.52ms and 0.37ms, and Chromium tools for capturing memory snapshots operate at a coarser granularity. Due to this practical limitation, we adopt the first approach and introduce mechanisms to preserve find-grained information about the functional dependencies and browser events. Our methodology comprises two phase and the algorithm is as follows:

- **Creating JS Capsules (Phase 1):** We climb the call-stack bottom-up and, for each function f(), inspect if its memory can be measured in isolation, i.e., the memory difference between the end and start of f() does not overlap with any subsequent function g() or events that are not f()'s dependent. If possible, we use f()'s total-time (i.e., start and end timestamps) for representing a distinct JS Capsule. Otherwise, we continue the search until we reach the top call (bar() in Figure 3a). In the case when such a measurement is not possible for the top call, we merge the timing information for consecutive function calls and encapsulate them in a single JS Capsule. However, in practice, we observed that this merging was only required for a small proportion of function.
- **Capturing information for JS Capsules (Phase 2):** We capture three pieces of information for a JS Capsule: memory, attributed events, and DOM API calls, the first quantifies memory while the latter two indicate the key browser and JS activity performed within the JS Capsule. The implementation details of extracting these features from the browser are discussed in the next section (Section 5).
  For memory, we take a difference of memory snapshots between the *end* and *start* timestamp of the respective JS Capsule. Each snapshot comprises the process's current memory and the allocator-wise memory distribution on a per-process basis.
  For browser events, we leverage a JS Capsule timing information to attribute events. Specifically, when a browser event is triggered as a response to a JS call (e.g., *recalculate style* in Figure 3a), one of three scenarios can happen: (i) the self-time for the event may overlap with f()'s total-time, (ii) the event may occur *after* f(), i.e., does not overlap with f()'s total-time, or (iii) the event may occur outside of renderer process context (e.g., GPU events) or main thread (chromium sometimes offload tasks for creating bitmaps of tiles to paint to the screen to raster/composite threads [74]). While the solution for the first scenario is straight-forward, i.e., events contained with the start and end timestamps are attributed to the respective JS Capsule, the other two present a challenge. Specifically, while the single-threaded browser mechanics for the main-thread ensures an ordering, some cascading events may occur outside of JS Capsule duration or in another browser thread/process (we call such events out-of-duration events and are mostly either *layout* or *paint*). For out-of-duration events, we adopt

[4]While total-time boundaries for the different f() may overlap with each other in Figure 3a, self-time boundaries are always non-overlapping due to single-threaded nature of JS execution.

a heuristic that is used in task attribution for leading web performance diagnosis tools such a LightHouse [1, 60]. The heuristic is based on the observation that renderer process emits *scheduling* tasks, e.g., *ScheduleStyleRecalculation* or *InvalidateLayout* etc, indicating which function f() is responsible for the subsequent out-of-duration events. We use this heuristic to attribute such events to the subset of preceding JS Capsules that occur after the last occurrence of the respective out-of-duration event and contain a respective scheduling task. For capturing JS DOM APIs, we instrument the website JS to log the DOM APIs called in a function.

Eliminating Noise: Note that garbage collection is non-deterministic and may have reclaimed memory during a JS Capsule's duration which will lead to discrepancies in memory measurements. To eliminate garbage collection-related non-determinism and keep measurements consistent, we force garbage collection *before* every snapshot. This calibration step minimizes the non-determinism of garbage collection call during a JS Capsule and is validated in Section 6.

JS Capsules structure: Each JS Capsule is represented as a key-value pair, mapping a custom unique ID for the JS Capsule to a set consisting of memory size, dependent functions, browser events and DOM APIs. We also log the dependency information between the functions, the DOM APIs and browser events on a per-function basis, i.e., a nested dictionary that maps JS Capsule-ID to f()-ID and f()-ID to DOM APIs and browser events. Since functions may have repetitive or anonymous names, we use the V8-generated unique IDs as f()-ID and store metadata such as the URL of the JS file that contained the f(), HTTP headers for the respective URL, and f() code content.

The JS Capsule-generation procedure is performed for each renderer process [5], since the single-thread browser mechanics for the main-thread are only limited to a single renderer process scope. The JS Capsules for a website are represented as a dictionary, mapping process-ID to the list of JS Capsule-IDs.

5 IMPLEMENTATION

In this section, we present the implementation details of our framework (Section 5.1), discuss our mobile phone-based testbed (Section 5.2), and present out JS classification techniques (Section 5.3) which we use for attribution.

5.1 Generating JS Capsules

The process of generating JS Capsules comprises: (i) loading a website with a mobile browser to generate logs, and (ii) processing these logs using the algorithm discussed in Section 4.3.

Capturing memory: We use Chromium's *requestMemoryDump* API [41] for capturing a continuous memory snapshot timeline[6]. Each snapshot comprises a process's current memory (*private footprint bytes* field) and allocator-wise distribution (*allocators* field), i.e., isolated timelines for Browser, GPU and each Renderer process. For each snapshot, we set *deterministic* to true in the API that enables deterministic results by forcing garbage collection [42].

Capturing JS functions and events: We use Chromium's *Tracing* API [40] to capture the low-level events during a page load. The Tracing API defines a number of different categories, among which *v8.cpu_profile* captures JS execution traces. For functions defined by the developer in HTML/JS files or referenced from included libraries, the *v8.cpu_profile* trace includes the JS timing information (*start* and *end* timestamps) and the metadata (function name, file URL, unique IDs assigned by V8, location of function code in file, etc.). Additionally, while generating the trace, we include all relevant event categories and capture cascading events. For example, we capture loading

[5]Multiple renderer processes may be spawned if the website has multiple iframes.
[6]Memory snapshots are collected every 50ms.

(*ParseHTML, FinishLoading* etc), scripting (*EvaluateScript, FunctionCall, v8.compile*, etc), rendering (*Invalidatelayout, Layout, Recalculatestyle*, etc), paint (*Paint, CompositeLayers, ImageDecode*, etc), among other categories. Specifically, we collect all the categories and events mentioned in Chrome's performance reference documentation [44].

Capturing DOM API calls: To track JS API calls, we wrote a generic JS framework that leverages JS *prototyping* [77] to track API calls and the DOM nodes that they interact with. At a high-level, this framework adds shims around all JS DOM API calls listed in [76] and tracks function arguments and the nodes that they manipulate. We inject this framework into a website's code using a proxy in our testbed and setup the proxy to communicate the results back to a data-collection server. For DOM-centered analysis, we use the Chrome *DOMSnapshot* API [39] which captures a webpage's entire DOM tree data-structure.

Log processing: Once the logs are collected, our Python-based framework processes the raw logs using the algorithm in Section 4.3 to extract JS Capsules. A key challenge in log processing is the lack of documentation available for log structure and their meaning. We perused Chromium's tracing code to understand the structure. The log-processing framework is implemented in 3.2K lines of python code. We plan to open-source the code to help the community in performance analysis.

5.2 Measurement Testbed

Our testbed comprises two Android phones and a dedicated desktop to automate any activity on phone (e.g., open or close browser, clearing app cache, etc) through *Android Debug Bridge (ADB)* [35]. To interact with Chromium APIs for log collection or load webpages, the testbed uses *chrome-remote-interface* [27] (implemented in *node.js*) and does not require device root access.

We collect measurements on two phones: LG G5 (high-end, 4GB RAM, 2x2.15, 2x1.6 GHz CPU, 5.3in screen, 554ppi resolution), and QMobile Q Infinity B (low-end, 1GB RAM, Quad-core 4x1.2 GHz CPU, 4.95in screen, 217ppi). The devices run the vendor's Android OS distribution (Android 7.0). Chrome v91 is used for our measurements, with no background applications or tabs to ensure that there is no background memory contention and all memory is available to the page load process. To prevent any cross-measurement contamination, a new browser instance is spawned and its application cache is cleared before each trial. The browser is terminated after each trial and each measurement is repeated five times. The phones are connected with an energy source to keep them always fully charged. We also ensure that the page fully loads by tracking the *onLoad* event.

To ensure reproducibility, the desktop hosts *mitmproxy* [30], a popular record-and-replay tool, to record webpages and later replays them to the mobile devices. The last-mile link (i.e., the device access link) is set to home networking conditions (100Mbps, 20ms RTT) during replay. A key advantage of using *mitmproxy* is its event-driven scripting ability to modify web objects on-the-go, i.e., a Python script can inspect the request/response objects at runtime and modify the response as required. We use this scripting ability to inject our JS API tracker framework. We conducted measurements for the Alexa top 1K websites (list collected in April, 2022). Our list provides good coverage over different website categories (inferred though *Cyren* tool [32]) such as *news (131 websites), shopping (47), social network (25), sports (13), search (41), business (165)* and *entertainment (87).* We only focus on the landing pages and do not interact with the webpage (e.g., scroll) to focus our measurements on only the base JS memory. Further, our devices load the webpages in their default language, i.e., there is no JS overhead of translating the webpages to English.

5.3 JS Classification

To aid in understanding the nature of JS Capsule functions, we classify the functions based on two sets of features: (i) *type*, i.e., origin, cross-origin, advertisements, trackers and libraries (e.g., jQuery, Ajax), and (ii) *functional*, i.e, browser events generated by a function.

Type (Source) classification: To classify if a function belongs to the same or cross-origin JS, we inspect the *Sec-Fetch-Site* field in the HTTP headers of the file containing the function. This field indicates the relationship between a request initiator's origin and the origin of the requested object [105]. For ads or trackers, we use a publicly available ad/tracker-blocking lists [92, 110] (collected in July, 2022) and classify the source's URLs by feeding the list to the *adblockparser* library [78] which uses regex matching for inference.

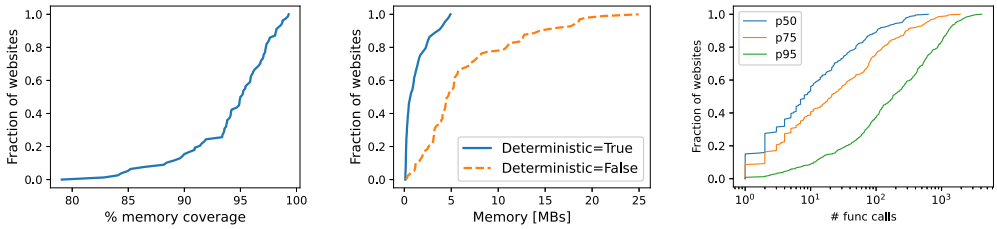
Contrary to origin, ad or tracker, classifying a function into libraries is more challenging. While the source URLs provide hints about the source, e.g., websites often fetch library code from 3rd-party servers and the URL includes the library name (such as *code.jquery.com/jquery-3.6.0.min.js*), there's no strict guarantee that this observation holds for all libraries and sites. Developers often host a library's code on their own servers and may even name the file in a different way. For instance, *drupal.org* uses standard libraries *jQuery* and *CreateJS* (used for animations) and the respective library code is included in a random file under *drupal.org/files/advagg_js/*. A simple URL-based matching is bound to miss such appearances and, thus, may not provide good coverage.

To tackle this challenge, we developed a database of all JS functions included our corpus of the top 1K websites. Taking inspiration from prior work [22], we generated unique fingerprints[7] for each function and inspected whether a function appears across multiple websites. We make the assumption that the appearance of the same function across multiple websites (we use a threshold of 5 websites) indicates that it is part of a shared library. While computing the frequency of a function across multiple websites, we ensure that we only use unrelated websites in our analysis. For example, we group together related websites, e.g., *google.com* and *google.co.uk* (or *apache.org* and *apachefriends.org*), and only use one in our frequency counts. We manually inspected the hostnames to mark such duplicate websites.

Functionality-based classification: Websites often comprise of tens of thousands of functions and understanding their impact on browser mechanics and the page load process can be an exhaustively tenuous tasks. The key intuition behind event-based classification is to understand how different functions differ in their cascading effects and, leveraging such classification, we can simplify the root-cause analysis by focusing on individual clusters, instead of individual functions or JS Capsules.

For this purpose, we leverage the event logs (discussed in Section 5.1) and represent each JS Capsule as a frequency histogram of browser events. We normalize the data and experimented with a number of clustering techniques including Kmeans, SVM and hierarchical clustering, and empirically evaluated Kmeans to be the best option. For the choice of "K", we tested a range of values between 2 to 500 and used the *elbow method* [96] for analyzing K's impact on sum of squared errors (SSE). We found the elbow to be K=113, representing the point after which the change in SSE smoothens and, consequently, we use 113 as the total number of clusters. Each cluster groups JS Capsules with similar frequency of browser events and provides us with a mechanism for understanding the similarity and differences between JS in different types of websites.

[7]While different text structural features or API features can be leveraged for computing fingerprint, we use *md5 hash* for the function code content due to its simplicity and fast computation.



(a) JS Capsule vs ground-truth JS memory.

(b) Impact of garbage collection on measurement noise.

(c) Number of functions per JS Capsule.

Fig. 4. Validation of JS Capsules: (A) Memory coverage of JS Capsules. (B) Garbage collection noise cancellation. (C) Completeness of JS Capsules

6 VALIDATION

6.1 Memory coverage and overhead

In order to validate memory coverage, we compared the sum of memory across all the JS Capsules in a website with its ground-truth JS memory. Since the existing tools are unable to isolate JS memory across various allocators, we calculate the ground truth by removing JS code from a webpage and taking the difference between the memory observed for the default webpage against the modified page with JS-removed [8]. Our key insight behind this methodology is that with JS removed, the allocators would not incur the memory overhead of executing JS (and cascading effects). Additionally, to verify that our logging techniques do not introduce overheads or noise into our memory measurements, we disable logging and snapshoting while computing ground-truth memory and use the *dumpsys* [38] for measuring the memory for the browser processes. We remove all the JS code from the website by emptying the JS files and *script* tags in HTML (both inline code and *src* field). Note that, we still keep the *script* tags in the DOM to keep the DOM structure intact. To calculate the total memory, we first aggregate the timelines of individual process memories (i.e., bin the *pss* values by time and take sum) and take the *max* of the aggregated timeline.

Figure 4a plots the memory coverage for 100 randomly selected websites. We are able to achieve >85% memory coverage for more than 90% of the websites – illustrating that we precisely capture memory and that our logging and snapshots do not incur significant noise. For the cases where coverage is below 95%, we observe excessive garbage collection is the key factor behind the coverage gap. Recall that, we force garbage collection before every measurement to eliminate non-determinism with memory measurements (Section 5.1). A consequence of this design choice is that the browser aggressively reclaims unused memory and this design choice results in our framework having slightly lower coverage values than the ground-truth. Moreover, the fact that our framework's memory coverage is within 100% shows that JS Capsule memory measurements do not include any additional overhead.

6.2 Measurement noise

In Figure 4b, we compare the total JS Capsule memory for 100 randomly selected websites with and without forced determinism (i.e., set deterministic field to control garbage collection before measurements). The figure plots the standard deviation for 10 repeated measurements and shows that our design choice highly reduces the noise in measurements and improves the accuracy of our framework.

[8]We validate JS removal by ensuring that no scripting related events are observed and the total JS scripting time is 0.

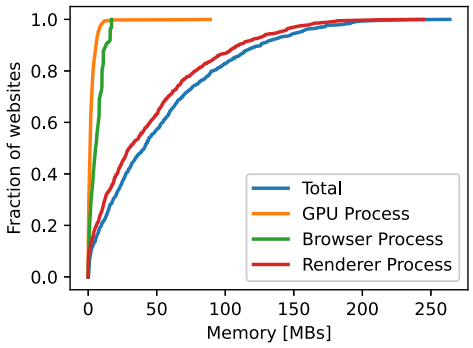


Fig. 5. Process-wise breakdown of JS memory across websites.

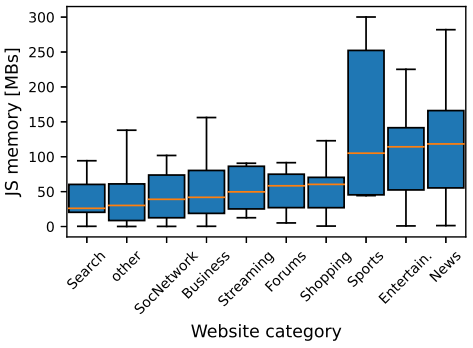


Fig. 6. JS memory distribution.

6.3 Library inference

To validate our library inference technique (Section 5.3), we generated a list of libraries which contribute atleast 5MB of memory to the top-20 websites (listed in Table 2). We randomly selected 50 of these libraries and manually inspected the function code and validated that our inferences are accurate.

6.4 JS Capsule distributions across websites

Our technique generates 103 capsules for the median website (213 at tail). To measure the number of JS functions encapsulated by a JS Capsule, we sort JS Capsules across a website based on the number of functions and Figure 4c plots the median, p75 and p95 number of JS Capsules in a website. Multiple functions per JS Capsule is expected as our design merges dependent functions into a single JS Capsule to tackle practical limitations, as discussed in Section 4.3. For the JS Capsules encapsulating a high number of functions, we observe that most functions (upto 80% at median) to be short-lived, taking less than 1ms to execute.

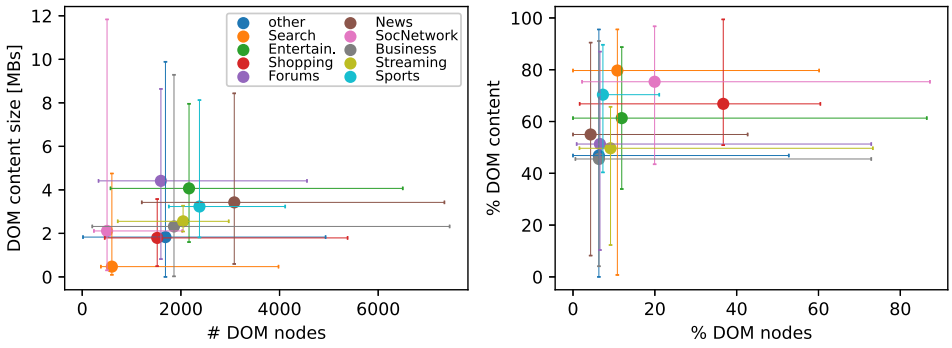
7 MEASUREMENT RESULTS

In this section, we present the memory results for the Alexa top-1K websites. Our goal is to quantify and characterize JS memory, and understand the root-cause of high JS memory usage.

7.1 JS Memory Across Websites

Figure 5 plots the distribution of JS memory across the 1K websites and breaksdown memory for the three browser processes. The Renderer's process contributes a significant amount of memory because of Chromium's architecture: the Renderer process performs the core activity required for loading a page and running JS (Section 2). Further, as the architecture defines a single Browser process that controls UI, network and tab switches, the low memory for the Browser process is by-design and minimizes the risk of a single tab exhausting Browser process memory and slowing down operations that also impact other tabs. Finally, the GPU process shows a significant memory footprint for only specific websites. We found a single library called *CreateJS* [31], used primarily for adding animations, to be the key reason behind GPU process' memory footprint.

Figure 5 shows that JS memory varies widely across the different websites, consuming none-to-minimal memory for some websites, while consuming >250MBs for others. This trend is due



(a) Number of DOM nodes and content size contributed by JavaScript.

(b) Percent of DOM nodes and content contributed by JavaScript.

Fig. 7. DOM and memory impact of JS across different website categories.
The bars represent the median value, with the error bars representing p5 and p95.

to the nature of these websites, since different websites vary in their extent of JS usage, content and functionality. As discussed in Section 2, developers typically define a basic DOM structure and content (e.g., text within *p*, image URL in *img*) in *index* HTML, which is later supplemented by JS DOM APIs [76] or libraries such as jQuery, e.g., adding new nodes by *appendChild()*, modifying text by *innerHTML*, changing node appearance by *style*, etc. In inspecting the DOM changes triggered by the JS Capsules, we observed that dynamic JS interactions with DOM vary widely across the websites. JS activity ranged from modifying styles, adding new nodes (e.g., 53 text nodes for median website, 439 at tail), adding content to nodes (e.g., 49KB of text at median, 578KB at tail) and adding event listeners (159 at median, 853 at tail).

To characterize the impact of JS on DOM, in Figures 7a and 7b, we present the DOM sizes and the percentage of DOM added by JS, while Figure 6 plots the JS memory for the websites in different categories. For simplicity, we characterize DOM along two axes: the number of nodes and the size of content (e.g., text, media) in each node. Unsurprisingly, DOM size is the heaviest for content-rich websites like *news*, *sports*, *entertainment* and *forums*. Surprisingly, client-side JS plays a significant role for such websites, with over 50% of content size added by JS at runtime. The trend in Figure 7b suggests that most websites, excluding *shopping* and the tail websites, define 80% or more of their DOM nodes statically in HTML and then heavily rely on client-side JS to add content to these nodes. In fact, the size of content added by JS correlates moderate-to-strong with the memory used by the website (Spearman's coefficient [79][9] of ρ=0.704 with p-value < 0.000), whereas it correlates weakly with the number of nodes (ρ=0.392 with p-value < 0.000). This correlation indicates that the JS memory of a website is expected to increase based on JS contribution's on populating and managing the DOM's content. Further, we observe this trend in Figure 6 where websites (e.g., *news* or *sports*) with heavier DOMs and higher contribution of JS manipulated DOM content tend to consume much more memory that comparatively simpler websites like *search* or *social networks*[10].

[9]We use Spearman correlation due to its ability to access monotonic relationships, while other coefficients like Pearson's are more suited for linear relationships [79].
[10]First-level social network pages are mostly simple login pages.

7.2 Characterizing JS Memory Allocations

In order to analyze the different sources of JS memory, Figure 8 plots the contribution of the different allocators. The figure contradicts a general notion held about JS memory that the footprint can be simply attributed to JS heap (e.g., arrays, compiled code, objects) and media objects (e.g., images, video). In fact, the median contribution by *V8* is capped at 27% and much more modest for *web cache* (responsible for storing media objects), highlighting that to understand mobile web memory one needs to examine beyond the *V8* and media objects.

The high footprint for *cc* (*compositor*) and *malloc*[11] reinforces the strong memory overhead of JS calls that impact the layout of DOM and webpage, resulting in rendering operations. In fact, the overhead is even higher that *V8* for the p75 and tail websites. In analyzing these websites, we observe that these websites tend to have complex DOM trees (i.e, number of nodes, tree depths and average number of children per node among the 75th percentile or higher). The indicates a strong relationship between the complexity of DOM tree layouts (i.e, number and type of nodes impacted) and the *cc* and *malloc* allocations (further discussed next).

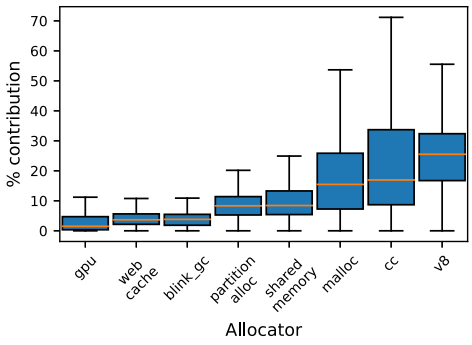


Fig. 8. Allocations responsible for JS memory.

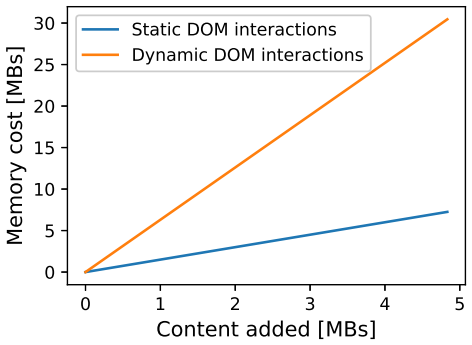


Fig. 9. Memory cost of static vs dynamic DOM interactions.

7.3 Takeaway

From our analysis in this section, we observe that JavaScript memory is a function of several key factors: First, the size of the content being manipulated in the DOM – this implies that web object size and format is a very promising method to address JavaScript memory. Second, most of the JavaScript-triggered allocations are outside of the JavaScript-heap (i.e., V8 heap) – which makes traditional tools miss leading. Our observations have implications both for how web pages are optimized but also in rethinking the tools used for analyzing JavaScript memory.

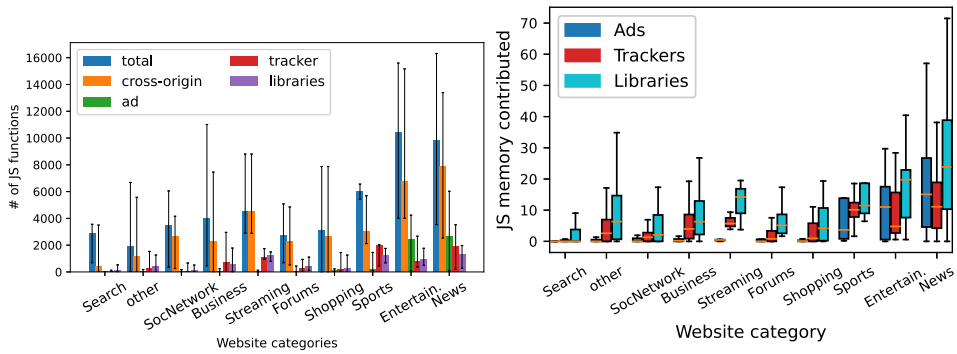
8 JAVASCRIPT CLASSIFICATION ANALYSIS

In this section, we leverage JS Capsules to analyze the nature of JavaScript (Section 8.1) and browser events that contribute the most memory (Section 8.2).

[11] *Malloc* is a general-purpose allocator and is used by different browser components. Our tests with synthetic webpages for different DOM sizes indicate that *malloc's* contribution increases with the DOM size, indicating that stores DOM metadata.

8.1 Impact of Ads, Trackers and Libraries

To get more visibility into the types of JS, we leverage the classification discussed in Section 5.3 to identify the source of memory, i.e., cross-origin, advertisement or ads, trackers, and general-purpose libraries. Figure 10a plots the breakdown of the number of unique functions executed for loading websites. We observe that *news* and *entertainment* websites execute upto 10,000 functions at the median. A majority of these functions are from cross-origin JS, which aligns with ancedotal knowledge that these websites often host their JS on CDNs due to their low-latency and faster fetch performance. Interestingly, ads and trackers JS show a significant presence for these website categories, contributing 4,543 and 3,227 functions at median, respectively. For context, this number is higher than the total number of functions for *shopping, search, forums* and *social network* categories. We observe a similar trend for the number of bytes in function code.



(a) Function breakdown. (b) Memory breakdown.

Fig. 10. Breakdown across different sources of JS.
The bars represent the median value, with the error bars representing p5 and p95.

To quantify the memory implications, in Figure 10b, we plot the memory contributed by such functions. We observe three key trends:

- First, across all the categories, library functions tend to contribute the highest amount of memory out of the three classes (i.e, ads, trackers and libraries). Table 2 compiles the list of top 20 such libraries and their primary purpose[12], ranked by memory. These libraries belong to multiple domains and provide a broad range of functionality from general functions (e.g., analytics, marketing) to APIs for browser interactions (e.g., DOM handling, networking requests, widgets and animations).
- Second, ads and trackers can result in a significant memory overhead, with over 17% cumulative impact for *entertainment* and over 20% for *news* and *sports*. We find the "number of ads" to be the key factor behind ad memory (*news* have as much as 7 ad iframes, compared to $\leq$4 for others). Interestingly, we observed that a tracker's memory footprint increases with the number of DOM nodes and event listeners, indicating that more complex websites tend to pay a higher memory cost for trackers.
- Third, and most interestingly, while there is a strong correlation between memory and number of functions, *the magnitude of the impact shows sharp contrast for libraries, as compared to ads or trackers JS.* This observation is presented in Table 3, where we calculate the correlation coefficient for the

[12]We manually analyzed the library documentation, if available, or code to understand their purpose. Few of the listed libraries were tracking-related and were not identified by the open-source tracker lists (Section 5.3) as such.

Library	Purpose	Memory (MBs)
s.ntv.io/serve/load.js	Analytics/tracker	25-32
widget.insent.ai	Widget	20-28
create.js	Animations	12-20.3
jquery	APIs for DOM manipulation, event handling & animations	5.2-17.1
cse.google.com	Programmable search engine	5.9-12.4
profitwell.com/js/profitwell.js	Marketing/subscription	7.3-9.7
hearstapps/...moapt-bidder-pb	Tracker	6.5-6.9
cloudfront.net/p.js	Analytics/tracker	4.4-7.3
ajax	Async network requests	3.45-7.3
static/vars hotjar.com	Analytics	3-7.8
static.scroll.com/js/scroll.js	Ad-free news browsing	2.2-14.6
apis.google.com/js/platform.js	Google auth	2-7.9
tinypass.com/xbuilder	Analytics	2.5-5.8
instream/video/client.js	Video ad integration	1.5-24.4
lightboxcdn.com/.../user.js	Analytics/tracker	0.7-10.3
edge.permutive.app/...-web.js	Ads & audience tracking	2.1-5.8
MessagingWithoutDetection.js	Messaging campaigns	2.9-5.3
jwplatform.com/libraries	Video player	1.9-7.8
sail-horizon.com/...spm.v1.min.js	Tracker	1.2-12.3
npttech.com/advertising.js	Ads & audience tracking	1.8-7

Table 2. Top libraries and their primary purpose, ranked by average memory usage.

JS memory	# of f()	Corr. coefficient (p-value)	Slope
All	All f()	0.898 (0.000)	0.00716
Ad-only	Ad f()	0.983 (0.000)	0.00520
Tracker-only	Tracker f()	0.936 (0.000)	0.00524
Library-only	Library f()	0.815 (0.000)	0.00892

Table 3. Correlation between memory and function distributions.

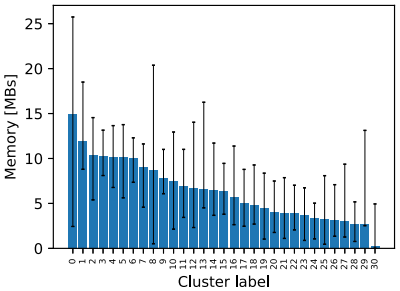


Fig. 11. Memory footprint for top 31 clusters.

memory vs #-of-function distributions. We further fit a regression model to the distribution and calculate the slope. High coefficient across the different breakdowns indicates a strong correlation: a website with a higher number of JS functions is expected to have a higher memory. However, the slope value, measuring the magnitude of increase, shows much stronger impact of library functions (~70% higher impact).

A key reason for this discrepancy lies in the actions performed by the JS functions across the three classes. Library functions actively interact with the main website DOM, while on the other hand, the scope and impact of ads and trackers are comparatively limited. Ads are generally displayed in iframes due to CORS [75] and, consequently, have their separate DOM attached to the main DOM tree. The scope of their JS, and extent of DOM manipulations, are thus limited to the iframe-DOM, which tend to be smaller and less-complex due to comparatively smaller size of ads, as compared to the main website's DOM. Trackers are expected to work transparently to the end-users and, while they may manipulate DOM by attaching event listeners to nodes, they do not alter the appearance and do not add any visual utility to the websites. We narrow our focus to the top-25 percentile JS Capsules (ranked by memory) and, comparing the impact of DOM for ads and library JS Capsules, we found that the median library function interactions with DOM touched 3.2X higher number of DOM nodes than the ad functions.

Takeaways: We observe that javascript for ads, libraries, and tracker generally consume significant memory. However, they consume memory for drastically different reasons which ties back to our observations about the relationship between DOM interactions and memory. On one hand, libraries interact with the main DOM which is generally richer and more complex, thus, they have a huge footprint. On the other hand, ads and trackers interact with independent and smaller DOM thus they have a samller footprint; however, pages generally contain many ads and trackers thus their cumulative footprints contributes a non-trivial fraction of a page's memory footprint.

Cluster	Top features
0	v8.compile(0.4), EvaluateScript(0.3), Layout(0.2)
1	EvaluateScript(0.4), v8.compile(0.2), Layout(0.2)
2	FrameCommittedInBrowser(0.5)
3	FunctionCall(0.5), Layout(0.3)
4	GPUTask(0.4)
5	Paint(0.2), {Animation, FunctionCall, GPUTask, InvalidateLayout, Layout, UpdateLayer}(0.1)
6	{ScheduleStyleRecalculation, UpdateLayoutTree}(0.2), {InvalidateLayout, Layout, Paint}(0.1)
7	Paint(0.4), {ScheduleStyleRecalculation, UpdateLayer}(0.1)
8	RequestAnimationFrame(0.5), CancelAnimationFrame(0.1)
9	{DecodeImage, DecodeLazyPixelRef, ImageDecodeTask, ImageUploadTask}(0.2)
10	XHRReadyStateChange(0.5), XHRLoad(0.3)
11	FrameCommittedInBrowser(0.2), {CompositeLayers, EvaluateScript, v8.compile}(0.1)
12	v8.parseOnBackground(0.4), {EvaluateScript, v8.compile}(0.1)
13	XHRReadyStateChange(0.2), {EvaluateScript, v8.compile, v8.parseOnBackground, XHRLoad}(0.1)
14	{EvaluateScript, v8.compile}(0.2), {CompositeLayers, UpdateLayer, v8.parseOnBackground}(0.1)
15	{InvalidateLayout, Layout}(0.2), {EvaluateScript, UpdateLayoutTree, v8.compile}(0.1)
16	ParseHTML(0.6)
17	{EvaluateScript, v8.compile}(0.2), v8.parseOnBackground(0.1)
18	Animation(0.2), {EvaluateScript, InvalidateLayout, Paint, UpdateLayer, v8.compile}(0.1)
19	FrameCommittedInBrowser(0.4), FunctionCall(0.1)
20	{DecodeLazyPixelRef, ImageDecodeTask, ImageUploadTask}(0.2), DecodeImage(0.1)
21	{XHRLoad, XHRReadyStateChange}(0.3)
22	v8.parseOnBackground(0.4), {v8.compile, EvaluateScript}(0.1)
23	XHRReadyStateChange(0.2), {EvaluateScript, FunctionCall, GPUTask, XHRLoad, v8.compile}(0.1)
24	FunctionCall(0.4), UpdateLayoutTree(0.1)
25	Paint(0.3), {InvalidateLayout, UpdateLayer}(0.1)
26	XHRReadyStateChange(0.3), XHRLoad(0.2), FunctionCall(0.1)
27	{DecodeImage, DecodeLazyPixelRef, ImageDecodeTask, ImageUploadTask}(0.2)
28	{Animation, DecodeImage, DecodeLazyPixelRef, ImageDecodeTask, ImageUploadTask, UpdateLayer, UpdateLayoutTree}(0.1)
29	{Animation, DecodeImage, DecodeLazyPixelRef, ImageDecodeTask, ImageUploadTask, Paint, UpdateLayer}(0.1)
30	UpdateLayer(0.7)

Table 4. Clusters' events composition. Color-coded based on the nature of events.

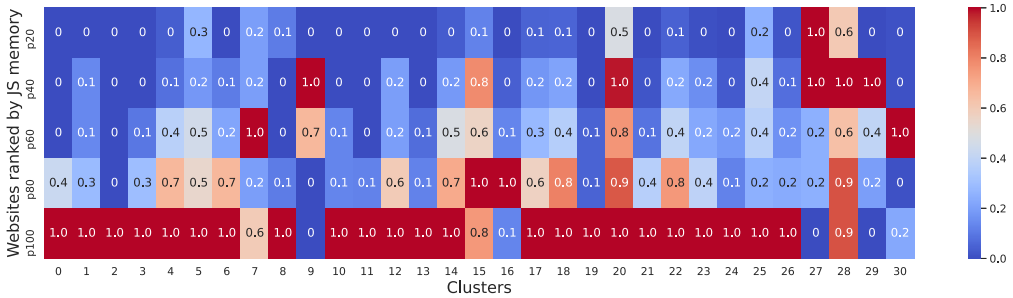


Fig. 12. Cluster appearance across different websites. The heatmap plots the number of times a cluster appears in a website JS, normalized by the max value.

8.2 Cluster Event Analysis

We conclude our measurements with an analysis of the JS cascading events and their memory impact, across websites with varying degree of JS memory. To identify the key event patterns that change across different types of websites, we leverage our previously discussed JS Capsule clustering. Since the Renderer process is the key contributor, we focus our analysis to the activity generated by the Renderer process.

We identified 113 clusters (using the approach discussed in Section 5.3) and observe that only 31 clusters have a median size of 1MB memory. We focus our discussions on these 31 clusters – Figure 11 plots the median memory per-cluster for these clusters.

To provide a semantic label for these clusters, ideally, we would simply analyze their constituent events; however, there are two challenges: (1) each cluster comprises 100s of event calls and (2)

there are many events with are present in every cluster, e.g., events such as *FunctionCall* and *EvaluateScript* show a strong presence in every cluster.

Thus, in order to determine what makes a cluster unique, and thus generate a semantic label, we apply a standard technique for calculating "feature importance" to our clusters. At a high level, the technique [94] assigns a 0-1 score to each feature, with a higher score representing higher importance. Table 4 compiles a list of such features (i.e., events) with at least 0.1 score for the 31 clusters. In order to distinguish between the events, we further color-code similar events together, e.g., red color for JS events *v8.compile, EvaluateScript, FunctionCall*, magenta for image-related events *DecodeImage, DecodeLazyPixelRef, ImageDecodeTask, ImageUploadTask*, etc.

In Figure 12, we divide the websites into five groups based on their JS memory footprint, and the heatmap visualizes how the presence of each cluster varies across the different groups of websites. Comparing the groups at the extremes, we observe a sharp contrast: the presence of most clusters gradually increases as we move from websites with low to high JS memory. This trend is primarily due to a higher reliance of memory-heavy websites on client-side JS, as shown in Figure 7. JS events, such as *v8.compile, EvaluateScript* and *FunctionCall*, are frequently observed for most cluster. A higher presence of these events directly indicates frequent JS evaluation and execution. Consequently, we also observe higher V8 memory for these clusters, as V8 requires memory for parsing code [99] and the overhead is expected to increase with the number of JS functions (or bytes of code) (Figure 10a). Similarly, layout and style related events, such as *Layout, ScheduleStyleRecalculation, InvalidateLayout* and *UpdateLayoutTree*, are frequently observed, highlighting that the top clusters involve rendering operations.

Interestingly, few of the clusters showed very peculiar activity patterns. We highlight two such patterns below:

- JS Capsules in cluster 8 involve *requestAnimationFrame()* API that is primarily used for updating animations [107]. Websites with these cluster heavily used JS for add dynamic animations that updated frequently over time, or waited for user-initiated interactions for animation updates. These JS Capsules also involved a heavy usage of timeout/interval functions. Interestingly, the memory for this cluster shows the widest variance (Figure 11) and we observed the timing and update intervals to be a key factor and a higher overhead is observed for a higher the frequency of updates.
- Clusters 9, 27, 28 and 29 heavily involved image related activity. Interestingly, these clusters, except 28, were observed more frequently for websites with lower JS memory overhead (Figure 12). A key reason why this pattern was observed is that the JS usage was limited to fetching images through Ajax APIs and adding the images to the DOM. Interestingly, we observed that the memory overhead of displaying the images varied with the format of images, being lowest for *webp* format. We further explored this observation though synthetic websites in our testbed and found that, for images with same number of pixels, different image formats incur different memory overhead owing to the compression and representation of images in the rendering engine. We observed webp to incur the lowest overhead followed by png (5% higher than webp), jpg (9.6% higher than webp) and svg (30.8% higher than webp).

Takeaway: We conclude from the cluster analysis that the JS cascading events show a sharp contrast across websites with different degrees of JS memory overhead. Websites with highest JS memory overhead tend to involve cascading events ranging from V8, rendering, paint, GPU and animations. Although significant work has been done in profiling the V8 interactions, the wide range of events motivate the need for profilers that can inspect other browser components in a similar fine-grained manner to shed more light into the browser mechanics behind JS memory overhead.

9 DISCUSSION

Our measurements indicate that the rampant use of client-side JS and its intricate relationship with browser and DOM mechanics is the key culprit behind JS memory overhead. In this section, we discuss the impact of our findings on several existing practical optimizations.

Mitigating Rendering Overhead: Since client-side rendering is a key contributor to memory, an obvious optimizations is to reduce the amount of client-side rendering. A number of optimizations are used today, ranging from Server-Side Rendering (i.e., server renders the DOM at request time and send the rendered DOM to the client), Pre-rendering (i.e., static DOM is generated at application build time), to Pre-rendering with Re-hydration [62, 88, 89]. These optimizations are expected to bring significant memory savings as they do not suffer from the high cost of building DOM through JS, and instead serve a complete or partially built DOM, that is later complemented by client-side JS (i.e., Re-hydration [62]). Our measurements with Opera Mini [85] (offloads page load to a proxy) shows up to 35-37% improvement at median for Alexa top-100. With Opera Mini, client browsers simply fetch a final, rendered markup document from a proxy server, which performs the heavy lifting to load the page. However, these optimizations often lead to privacy concerns [55] and reduced interactivity, as client-side JS is completely or partially eliminated or requires network interactions with the proxies.

Recent works [55, 83, 102] have explored automatic solutions for reducing client-side overhead by leveraging split-browser solutions or caching and reusing expensive JS computations. We expect these solutions to have promising impact on memory overhead. However, correctly identifying the subset of JS to either offload or cache remains an open challenge and requires find-grained understanding of JS functions impact on webpage functionality and utility.

Lazy Rendering: One interesting aspect with rendering is that the browser generates the render representation for the entire webpage and then gradually paints it on screen, based on the user's viewport [47, 58]. Mobile websites, especially news, tend to be vertically long with only a small proportion of content visible in a viewport. However, the browser still needs to keep the render-related data in-memory for the whole webpage. We leveraged JS *MutationObserver* for controlling the render behavior to emulate a scenario where the browser only renders the above-the-fold content. At a high-level, we inspected the position of a visible DOM node on screen and disables rendering for the nodes that are outside above-the-fold. To disable rendering, we set the *display:none* CSS property for the respective node that forces the browser to skip the node during render tree construction and the node plays no part during the rendering phase. Although we observed modest improvements in memory (7%) for the median website, tail websites showed upto 25% reduced memory usage. Most of the websites where this *lazy render* scheme resulted in >10% savings included news websites (e.g., dawn.com).

While our strawman approach to limit rendering to a subset of nodes may not be ideal, the results show that there is a potential to reduce memory overhead. However, an ideal solution for such an approach should be browser-based and should tune the rendering mechanics. While browsers include abstractions like *Shadow DOM* [106] for compartmentalizing DOM trees, the key challenge towards implementing such a strategy lies in introducing browser abstractions through which render-tree can be compartmentalized and lazily loaded-based on the DOM nodes coordinates on screen.

Dead-code Elimination and De-bloating: A number of works focus on removing unused or non-critical JS code [48, 49, 108]. In addition to debloating, research works [21–23] have also explored replacing JS with HTML whenever possible However, the HTML-replacement part is only valid for JavaScript whose execution is not expected to change based on client-side state, e.g. cookies, device-type etc. While these approaches are expected to save V8 memory as the

amount of code required to parse is reduced, they are expected to keep the DOM tree content and manipulations intact (at best, only replace a subset of JS manipulations), and are not expected to result in significant savings for JS indirect memory overhead. Further, removing non-critical JS (e.g., ads/trackers) are only expected to save significant memory for a narrow subset of websites (Figure 10a).

10 RELATED WORK

Measurement Studies: Recent works [19, 33] explored energy usage and impact of device resources on QoE for low-end devices. [6] discussed mobile phones characteristics in different regions of the world, while [2] quantifies memory resources for devices in different parts of the world and analyzes the implications of low-end device memory on performance. Our work differs from these related works in several key ways. In contrast to our prior work [2], our primary focus is specifically on JS memory, and thus we introduced the design and implementation of a novel tool (JS Capsule). Additionally, we advance on our prior work [2] by leveraging JS Capsule and other tools to perform fine-grain measurements which allows us to directly discuss the memory overhead in the context of browser, especially the rendering mechanics and DOM interactions. In contrast to [33], we perform a more thorough investigation how much and why is the memory allocated during page loads. While [19] focuses on browser events in the context of energy usage, our measurements are for a completely different domain and our measurement technique goes a step further by attributing browser events to JS functions.

Web JavaScript Optimizations: Prior work on optimizing JS ranges from analysis of V8 engine [10, 52, 95, 99, 109], reducing JS computational bottlenecks [82, 83, 85, 102], to debloating webpages by removing excess JS [21–23, 48, 49, 108]. A final class of optimizations (e.g., Google AMP [64]) focus on improving web performance by limiting JS execution to ensure that JS executions are non-blocking the rendering of main page. In contrast, our primary goal in this paper is to understand memory overhead of JS in the current web ecosystem. For this purpose, we developed a novel measurement technique and conducted a study for Alexa top 1K pages to quantify and characterize JS memory and its implications on browser mechanics.

Browser Optimizations: Recent works have proposed several browser optimizations to improve performance and resource usage. [99] discusses lazy parsing schemes for V8 to improve computational and memory usage. [13] discusses heuristic-based interventions in Chrome to disable advertisement JS. Similarly, others have focused on ad and tracker blocking schemes for browsers [91]. The context of our work stands in stark contrast to these works. First, instead of just focusing on V8, our study analyzes the cross-component aspects of JS and discusses the implications of direct (i.e., V8) and indirect memory overhead. Instead of focusing on a particular type of JS such as advertisements, we analyze all sources of JS in a website.

11 CONCLUSION

Although JavaScript memory is a crucial aspect of the mobile web page load process, we lack sufficiently fine-grained and precise tooling or methodology for accurately and effectively capturing JavaScript memory. In this paper, we develop a novel framework, JS Capsules, that builds on knowledge of common browser architectures and targetted logging information to capture fine-grained and precise JavaScript memory measurements. Using our tool, we analyzed 1000 websites and report on the common patterns and discuss the impact of our findings on several practical optimizations. Our measurements show that JavaScript results in significant indirect memory overheads (i.e., non-V8 memory) and a wide range of browser events, especially rendering, contribute to the overhead.

While significant amount of work has been undertaken to develop profilers for V8, our measurements indicate that there is a need for holistic profilers for other browser components, such as compositor, that provide an in-depth understanding of JavaScript APIs interactions with such components. Most importantly, our measurements also highlight a need for fine-grained models, benchmarks, and statistical techniques to adequately reason about the finer level of granularity that our tool provides.

12 ACKNOWLEDGMENTS

We thank the anonymous reviewers and our shepherd, Mike Ferdman, for their invaluable comments. This work is supported by NSF grants CNS-1814285 and a Google Faculty Award.

REFERENCES

[1] Lighthouse | tools for web developers | google developers.
[2] Reference removed for double-blind review.
[3] These were the 5 most popular phone brands in africa in 2018. bit.ly/30dI2OT.
[4] A low-profile, chinese handset maker has taken over africa's mobile market. bit.ly/2W1wAGW.
[5] V. Agababov, M. Buettner, V. Chudnovsky, M. Cogan, B. Greenstein, S. McDaniel, M. Piatek, C. Scott, M. Welsh, and B. Yin. Flywheel: Google's data compression proxy for the mobile web. In *12th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 15)*, pages 367–380, 2015.
[6] S. Ahmad, A. L. Haamid, Z. A. Qazi, Z. Zhou, T. Benson, and I. A. Qazi. A view from the other side: Understanding mobile phone characteristics in the developing world. page 319–325, 2016.
[7] android.com. Android (go edition). powering entry-level devices., 2020.
[8] H. Archive. State of the web.
[9] I. Archive. Wayback machine.
[10] K. Basques. Fix memory problems, 2020.
[11] A. Bhattacharya. Internet use in india proves desktops are only for westerners, 2017.
[12] Z. S. Bischof, J. P. Rula, and F. E. Bustamante. In and out of cuba: Characterizing cuba's connectivity. In *Proceedings of the 2015 Internet Measurement Conference*, pages 487–493, 2015.
[13] C. Blog. Protecting against resource-heavy ads in chrome. https://blog.chromium.org/2020/05/resource-heavy-ads-in-chrome.html.
[14] The chinese unknown that's making africa's phones. bit.ly/2VUl5kB.
[15] J. W. Bojan Pavic, Chris Anstey. Why does speed matter? https://web.dev/why-speed-matters/.
[16] J. Brutlag. Speed matters for google web search, 2009.
[17] M. Butkiewicz, H. V. Madhyastha, and V. Sekar. Understanding website complexity: measurements, metrics, and implications. In *Proceedings of the 2011 ACM SIGCOMM conference on Internet measurement conference*, pages 313–328, 2011.
[18] M. Butkiewicz, D. Wang, Z. Wu, H. Madhyastha, and V. Sekar. Klotski: Reprioritizing web content to improve user experience on mobile devices. In *NSDI*, 2015.
[19] Y. Cao, J. Nejati, M. Wajahat, A. Balasubramanian, and A. Gandhi. Deconstructing the energy consumption of the mobile page load. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 1(1):1–25, 2017.
[20] J. C. Chang, N. Hahn, Y. Kim, J. Coupland, B. Breneisen, H. S. Kim, J. Hwong, and A. Kittur. When the tab comes due: challenges in the cost structure of browser tab usage. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*, pages 1–15, 2021.
[21] M. Chaqfeh, R. Coke, J. Hu, W. Hashmi, L. Subramanian, T. Rahwan, and Y. Zaki. Jsanalyzer: A web developer tool for simplifying mobile web pages through non-critical javascript elimination. *ACM Transactions on the Web (TWEB)*.
[22] M. Chaqfeh, M. Haseeb, W. Hashmi, P. Inshuti, M. Ramesh, M. Varvello, F. Zaffar, L. Subramanian, and Y. Zaki. To block or not to block: Accelerating mobile web pages on-the-fly through javascript classification. *arXiv preprint arXiv:2106.13764*, 2021.
[23] M. Chaqfeh, Y. Zaki, J. Hu, and L. Subramanian. Jscleaner: De-cluttering mobile webpages through javascript cleanup. In *Proceedings of The Web Conference 2020*, pages 763–773, 2020.
[24] chromium.org. Multi-process architecture. https://www.chromium.org/developers/design-documents/multi-process-architecture/.
[25] C. Cimpanu. All the chromium-based browsers. https://www.zdnet.com/pictures/all-the-chromium-based-browsers/12/.

[26] J. Clement. Percentage of mobile device website traffic worldwide from 1st quarter 2015 to 2nd quarter 2022. https://www.statista.com/statistics/277125/share-of-website-traffic-coming-from-mobile-devices/.
[27] M. contributors. chrome-remote-interface. https://github.com/cyrus-and/chrome-remote-interface.
[28] M. contributors. Css object model (cssom), 2020.
[29] M. contributors. Introduction to dom, 2020.
[30] A. Cortesi, M. Hils, T. Kriechbaumer, and contributors. mitmproxy: A free and open source interactive HTTPS proxy, 2010. [Version 6.0].
[31] createjs.com. Createjs. https://createjs.com/.
[32] cyren.com. Cyren website url category checker. https://www.cyren.com/security-center/url-category-check.
[33] M. Dasari, S. Vargas, A. Bhattacharya, A. Balasubramanian, S. R. Das, and M. Ferdman. Impact of device performance on mobile internet qoe. In *Proceedings of the Internet Measurement Conference 2018*, pages 1–7. ACM, 2018.
[34] developer.android.com. Activitymanager.memoryinfo. https://developer.android.com/reference/android/app/ActivityManager.MemoryInfo.
[35] A. Developers. Android debug bridge (adb). https://developer.android.com/studio/command-line/adb.
[36] A. Developers. Memory allocation among processes.
[37] A. Developers. Overview of memory management. https://developer.android.com/topic/performance/memory-overview.
[38] A. Developers. Dumpsys, 2020.
[39] C. Developers. Chrome devtools protocol - domsnapshot domain. https://chromedevtools.github.io/devtools-protocol/tot/Tracing/#type-MemoryDumpConfig.
[40] C. Developers. Chrome devtools protocol - tracing domain. https://chromedevtools.github.io/devtools-protocol/tot/Tracing/.
[41] C. Developers. Chrome devtools protocol - tracing domain. https://chromedevtools.github.io/devtools-protocol/tot/Tracing/#type-MemoryDumpConfig.
[42] C. Developers. Chrome devtools protocol - tracing domain. https://chromedevtools.github.io/devtools-protocol/tot/Tracing/#method-requestMemoryDump.
[43] C. developers. Memoryinfra. https://chromium.googlesource.com/chromium/src/+/master/docs/memory-infra/.
[44] C. Developers. Timeline event reference. https://developer.chrome.com/docs/devtools/evaluate-performance/performance-reference/.
[45] developers.android.com. Logcat command-line tool. https://developer.android.com/studio/command-line/logcat.
[46] D. Digital. Milliseconds make millions: A study on how improvements in mobile site speed positively affect a brand's bottom line. https://bit.ly/3rpm8WP.
[47] H. Djirdeh. Apply instant loading with the prpl pattern.
[48] H. Djirdeh. Reduce javascript payloads with code splitting.
[49] H. Djirdeh. Reduce javascript payloads with code splitting.
[50] docs.oracle.com. 10 garbage-first garbage collector tuning. https://docs.oracle.com/javase/9/gctuning/garbage-first-garbage-collector-tuning.htm.
[51] N. Dukkipati, T. Refice, Y. Cheng, J. Chu, T. Herbert, A. Agarwal, A. Jain, and N. Sutin. An argument for increasing tcp's initial congestion window. *Computer Communication Review*, 40(3):26–33, 2010.
[52] J. et al. Visiblev8: In-browser monitoring of javascript in the wild. In *IMC*, 2019.
[53] Facebook. Facebook lite, 2020.
[54] geeksforgeeks.org. Why javascript is a single-thread language that can be non-blocking? https://www.geeksforgeeks.org/why-javascript-is-a-single-thread-language-that-can-be-non-blocking/.
[55] A. Goel, V. Ruamviboonsuk, R. Netravali, and H. V. Madhyastha. Rethinking client-side caching for the mobile web. In *Proceedings of the 22nd International Workshop on Mobile Computing Systems and Applications*, pages 112–118, 2021.
[56] GSMA. The mobile economy. sub-saharan africa 2020, 2020.
[57] R. Hat. Interpreting /proc/meminfo and free output for red hat enterprise linux.
[58] K. Hempenius. Use lazysizes to lazy-load images.
[59] T. Hoff. Latency is everywhere and it costs you sales - how to crush it. http://highscalability.com/latency-everywhere-and-it-costs-you-sales-how-crush-it.
[60] P. Hulce. Long task attribution in lighthouse. https://docs.google.com/presentation/d/1sslnZi2MYyKlNb6LoS_vTlcvshkUR0IUekacq4mZMhs.
[61] U. T. Inc. Uber lite, 2020.
[62] A. O. Jason Miller. Rendering on the web.
[63] L. Johnson. 47 percent of top retailers have a mobile site and app: study. https://www.retaildive.com/ex/mobilecommercedaily/47-percent-of-top-retailers-have-a-mobile-site-and-app-study.

[64] B. Jun, F. E. Bustamante, S. Y. Whang, and Z. S. Bischof. Amp up your mobile web experience: Characterizing the impact of google's accelerated mobile project. In *The 25th Annual International Conference on Mobile Computing and Networking*, pages 1–14, 2019.

[65] M. Kearney. Memory terminology.

[66] M. Kearney. Record heap snapshots.

[67] M. Kearney and K. Basques. Analyze runtime performance, 2020.

[68] kernel.org. Out of memory management. https://www.kernel.org/doc/gorman/html/understand/understand016.html.

[69] R. Ko, J. Mickens, B. Loring, and R. Netravali. Oblique: Accelerating page loads using symbolic execution. In *18th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 21)*, pages 289–302, 2021.

[70] M. Kosaka. Inside look at modern web browser (part 1). https://developer.chrome.com/blog/inside-browser-part1/.

[71] M. Kosaka. Inside look at modern web browser (part 1). https://developer.chrome.com/blog/inside-browser-part1/#site-isolation.

[72] J. Kupoluyi, M. Chaqfeh, M. Varvello, R. Coke, W. Hashmi, L. Subramanian, and Y. Zaki. Assessing the impact of javascript dead code elimination on mobile web performance. In *ACM Internet Measurement Conference, IMC 2022, Nice, France, October 25-27, 2022*, page (Just Accepted), 2022.

[73] P. Lewis. Avoid large, complex layouts and layout thrashing. https://developers.google.com/web/fundamentals/performance/rendering/avoid-large-complex-layouts-and-layout-thrashing.

[74] S. Mardani, A. Goel, R. Ko, H. V. Madhyastha, and R. Netravali. Horcrux: Automatic javascript parallelism for resource-efficient web computation. In *15th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 21)*, 2021.

[75] MDN. Cross-origin resource sharing (cors).

[76] MDN. Manipulating documents. https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Client-side_web_APIs/Manipulating_documents.

[77] MDN. Object prototypes.

[78] L. M. m. Mikhail Korobov, Marcin Lyko. adblockparser. https://github.com/scrapinghub/adblockparser.

[79] L. Myers and M. J. Sirois. Spearman correlation coefficients, differences between. *Encyclopedia of statistical sciences*, 12, 2004.

[80] U. Naseer and T. Benson. Configtron: Tackling network diversity with heterogeneous configurations. In *9th {USENIX} Workshop on Hot Topics in Cloud Computing (HotCloud 17)*, 2017.

[81] J. Nejati and A. Balasubramanian. An in-depth study of mobile browser performance. In *Proceedings of the 25th International Conference on World Wide Web*, pages 1305–1315, 2016.

[82] R. Netravali and J. Mickens. Prophecy: Accelerating mobile page loads using final-state write logs. In *NSDI*, 2018.

[83] R. Netravali, A. Sivaraman, J. Mickens, and H. Balakrishnan. Watchtower: Fast, secure mobile page loads using remote dependency resolution.

[84] X. Nie, Y. Zhao, G. Chen, K. Sui, Y. Chen, D. Pei, M. Zhang, and J. Zhang. Tcp wise: One initial congestion window is not enough. In *Performance Computing and Communications Conference (IPCCC), 2017 IEEE 36th International*, pages 1–8. IEEE, 2017.

[85] Opera. Opera mini, 2020.

[86] oreilly.com. Bing and google agree: Slow pages lose users. http://radar.oreilly.com/2009/06/bing-and-google-agree-slow-pag.html.

[87] A. Osmani. Adaptive loading - improving web performance on low-end devices.

[88] A. Osmani. The cost of javascript in 2019.

[89] A. Osmani. Loading web pages fast on a dollar 20 feature phone.

[90] perfetto.dev. Debugging memory usage on android. https://perfetto.dev/docs/case-studies/memory.

[91] B. Pourghassemi, J. Bonecutter, Z. Li, and A. Chandramowlishwaran. adperf: Characterizing the performance of third-party ads. *arXiv preprint arXiv:2002.05666*, 2020.

[92] M. Ryanbr and Khrin. Easylist / easyprivacy / fanboy lists. https://github.com/easylist/easylist/.

[93] S. S. Web performance risks: spotlight on javascript vs. low-end mobiles, 2020.

[94] M. Saarela and S. Jauhiainen. Comparison of feature importance measures as explanations for classification models. *SN Applied Sciences*, 3(2):1–12, 2021.

[95] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song. A symbolic execution framework for javascript. In *2010 IEEE Symposium on Security and Privacy*, pages 513–528. IEEE, 2010.

[96] P. Sinha. Determine the optimal value of k in k-means clustering. https://www.geeksforgeeks.org/ml-determine-the-optimal-value-of-k-in-k-means-clustering/.

[97] statcounter.com. Mobile browser market share worldwide. https://gs.statcounter.com/browser-market-share/mobile/worldwide.

[98] V. K. Tom Wiltzius and the Chrome Graphics team. Gpu accelerated compositing in chrome. https://www.chromium.org/developers/design-documents/gpu-accelerated-compositing-in-chrome.
[99] M. H. Toon Verwaest. Blazingly fast parsing, part 2: lazy parsing. https://v8.dev/blog/preparser.
[100] V8. V8 javascript engine. https://v8.dev/.
[101] X. Wang, A. Balasubramanian, A. Krishnamurthy, and D. Wetherall. Demystifying page load performance with wprof. In *NSDI*, 2013.
[102] X. S. Wang, A. Krishnamurthy, and D. Wetherall. Speeding up web page loads with shandian. In *13th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 16)*, pages 109–122, 2016.
[103] R. Waugh. There are now more mobile phones on the planet than there are people.
[104] M. web docs. Call stack. https://developer.mozilla.org/en-US/docs/Glossary/Call_stack.
[105] M. web docs. Sec-fetch-site. https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Sec-Fetch-Site.
[106] M. web docs. Using shadow dom. https://developer.mozilla.org/en-US/docs/Web/Web_Components/Using_shadow_DOM.
[107] M. web docs. Window.requestanimationframe(). https://developer.mozilla.org/en-US/docs/Web/API/window/requestAnimationFrame.
[108] webpack. Tree shaking. https://webpack.js.org/guides/tree-shaking/.
[109] W. Xu, F. Zhang, and S. Zhu. Jstill: mostly static detection of obfuscated malicious javascript code. In *Proceedings of the third ACM conference on Data and application security and privacy*, pages 117–128, 2013.
[110] yoyo.org. Blocking with ad server and tracking server hostnames. https://pgl.yoyo.org/adservers/.
[111] T. Zimmermann, J. Rüth, B. Wolters, and O. Hohlfeld. How http/2 pushes the web: An empirical study of http/2 server push. In *2017 IFIP Networking Conference (IFIP Networking) and Workshops*, pages 1–9. IEEE, 2017.

Received October 2022; revised December 2022; accepted January 2023