Horcrux: Automatic JavaScript Parallelism for Resource-Efficient Web Computation

Shaghayegh Mardani[1], Ayush Goel[2], Ronny Ko[3], Harsha V. Madhyastha[2], Ravi Netravali[4]
[1]*UCLA*, [2]*University of Michigan*, [3]*Harvard University*, [4]*Princeton University*

Abstract

Web pages today commonly include large amounts of JavaScript code in order to offer users a dynamic experience. These scripts often make pages slow to load, partly due to a fundamental inefficiency in how browsers process JavaScript content: browsers make it easy for web developers to reason about page state by serially executing all scripts on any frame in a page, but as a result, fail to leverage the multiple CPU cores that are readily available even on low-end phones.

In this paper, we show how to address this inefficiency without requiring pages to be rewritten or browsers to be modified. The key to our solution, Horcrux, is to account for the non-determinism intrinsic to web page loads and the constraints placed by the browser's API for parallelism. Horcrux-compliant web servers perform offline analysis of all the JavaScript code on any frame they serve to conservatively identify, for every JavaScript function, the union of the page state that the function could access across all loads of that page. Horcrux's JavaScript scheduler then uses this information to judiciously parallelize JavaScript execution on the client-side so that the end-state is identical to that of a serial execution, while minimizing coordination and offloading overheads. Across a wide range of pages, phones, and mobile networks covering web workloads in both developed and emerging regions, Horcrux reduces median browser computation delays by 31-44% and page load times by 18-37%.

1 Introduction

Despite accounting for over half of all global web traffic [28, 30, 76], mobile browsing in the wild continues to operate far slower than what users can endure [17, 18, 34], with page loads often taking upwards of 10 seconds [14, 79]. Since users are more likely to abandon pages that are slow to load [39], the current sub-optimal state of mobile web performance negatively impacts not only user experience, but also the revenue of content providers [31].

A key contributor to slow page loads on mobile devices is the computation that browsers must perform to load a page [85, 62, 64, 22], most of which is accounted for by the execution of JavaScript code (§2). Numerous solutions have attempted to reduce the amount of necessary client-side computation, either by requiring developers to manually rewrite pages [37] or by having clients offload page load computations to more powerful servers [68, 67, 71, 13, 64, 32]. However, solutions of the former class come at the expense of manual effort and page functionality, while those in the latter class are largely unviable in practice (§7). Offloading to proxies [68, 67, 71] is infeasible in today's HTTPS-by-default web, while systems [64] in which origin servers return post-processed pages that elide computations risk compromising correctness since servers lack visibility into client-side state (e.g., localStorage) that can affect control flow in a page load.

We pursue an alternative and complementary approach: instead of attempting to *reduce* the amount of computation that web clients must perform, we seek to execute the necessary computation on client devices *more efficiently*. In particular, we observe that there exists a fundamental inefficiency in the computation model that browsers employ (§2). To simplify page development, JavaScript execution is single-threaded [65, 64], and worse, JavaScript and rendering tasks are forced to share a single "main" thread per frame in a page [38]. Consequently, browsers are unable to take advantage of the growing number of CPU cores available on popular phones in both developed and emerging regions [24, 25]. This inefficiency will only worsen as, due to energy constraints, increased core counts have become the main source of compute resource improvements on phones [77, 35].

A natural solution to this inefficiency is to parallelize JavaScript computations across a device's available cores. Browsers have included support for pages to spin up parallel JavaScript computation threads in the form of Web Workers [55, 2] for over 8 years now. Yet, only a handful of the top 1,000 sites use Workers on their landing pages, largely due to the challenges of writing efficient, concurrent code [15, 45]. These challenges manifest in two ways for the web.

- Determining *which* JavaScript executions on a page frame can be safely parallelized requires a precise understanding of the page state accessed by every script, due to the language's lack of synchronization mechanisms (e.g., locks). Placing the onus of this task on web developers [53, 6] is impractical, while reliance on browsers to speculatively make parallelism decisions [21, 56] is inefficient (§6.3).
- *How* to efficiently execute scripts in parallel is also not straightforward due to the restrictions that browsers impose on Workers. In particular, they cannot access a page's JavaScript heap or DOM tree, and coordinating with the main thread, which has these privileges, adds overheads.

Our goal is to automatically parallelize JavaScript computations for *legacy pages* on *unmodified browsers*, thereby addressing the cognitive and operational overheads involved in explicitly making parallelism decisions. Our solution, **Horcrux**, achieves these goals by employing a judicious split between clients and servers, hence preserving HTTPS' end-to-end content integrity and privacy guarantees [67]. Servers perform the heavy lifting of finding parallelism opportunities and embed that information in their pages. Clients then run a JavaScript scheduler that efficiently manages parallelism using runtime information that servers lack, i.e., number of available threads, control flows taken in the current load. Three primary insights guide our design of Horcrux.

First, we ensure that any introduced parallelism preserves the final page state that developers expected when they wrote the page. For this, Horcrux forces computations that exhibit state dependencies (e.g., a read-write dependency on a global variable) to run serially in an order that matches the legacy load, while allowing other computations to run in parallel. Key to enabling this is Horcrux's offline use of concolic execution [48, 36, 80] on servers to explore all possible control flows on a page and identify all state that each JavaScript function *might* access, irrespective of how client-side nondeterminism could affect any particular load.

Second, Horcrux minimizes client-side coordination overheads by carefully partitioning responsibilities between the main browser thread and the Web Workers it spawns. Horcrux reserves the main thread for coordinating Worker offloads, managing global page state, and running DOM computations (which Workers cannot); all other computations are offloaded. This yields two benefits. First, scripts typically interleave computations that can and cannot be offloaded; Horcrux maximally parallelizes the former while carefully mediating the latter. Second, by keeping the main thread largely idle, Horcrux quickly adapts the parallelization schedule to the runtimes of JavaScript computations, offloading the next computation as soon as a Worker becomes available.

Third, the granularity at which Horcrux parallelizes JavaScript execution is crucial with respect to overheads and potential parallelism. A natural solution would be to offload the invocations of JavaScript functions, which account for 94% of JavaScript source code. However, the sheer number of invocations in a typical page load makes this too costly. Instead, we observe that functions are typically invoked hierarchically (i.e., nested functions), with significant state sharing within a hierarchy, but less across them. Therefore, Horcrux offloads at the granularity of root function invocations, or the root of each hierarchy along with its nested constituents. Compared to per-function offloading, this requires 4× fewer offloads while achieving 73% of the potential speedup.

We evaluated Horcrux using over 650 diverse pages, live mobile networks (LTE and WiFi), and three phones, that collectively represent browsing scenarios in both developed and emerging regions. Our experiments across these conditions reveal that Horcrux reduces median browser computation delays by 31-44% (0.9-1.5 secs), which translates to page load time and Speed Index speedups of 18-29% and 24-37%, respectively. Further, Horcrux's median benefits are 1.3-2.1× larger than prior compute-focused web accelerators, and 1.4-2.1× more than (complementary) network optimizations.

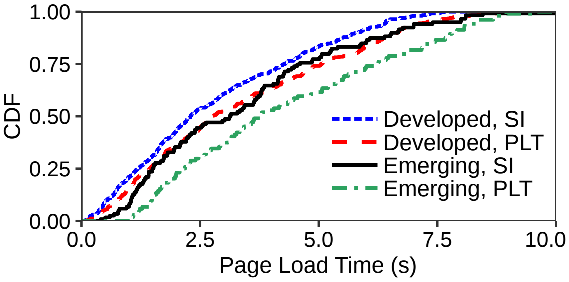


Figure 1: **Load times often exceed user tolerance levels (3 seconds) even when all network delays are removed.**

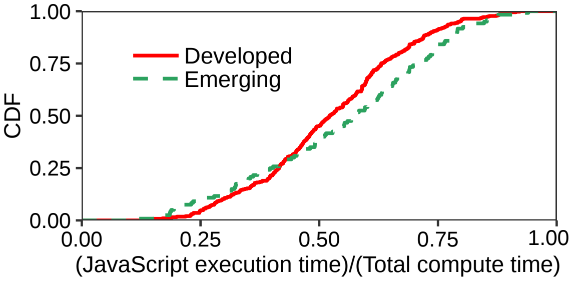


Figure 2: **JavaScript's role in browser computation delays.**

Taken together, our results highlight that, despite being written for a serial browser computation model, existing pages are surprisingly ripe with parallelization opportunities. Horcrux shows how such opportunities can be exploited under the hood, without having developers manually rewrite their pages. Source code and datasets for Horcrux are available at https://github.com/ShaghayeghMrdn/horcrux-osdi21.

2 Motivation and Background

Numerous studies have reported that client-side (browser) computation is a significant contributor to poor mobile web performance [85, 62, 79, 64]. We reproduce this finding below (§2.1), present measurements to elucidate why such delays are so pronounced (§2.1), and trace the origins for this poor performance to the computation model used by browsers today (§2.2). Our experimental setup (§6.1) covers web workloads in both developed and emerging markets by considering popular pages in the US and Pakistan and loading those pages on common phones in each region. Pages in the emerging region generally involve less JavaScript code, but are loaded on phones with fewer compute resources.

2.1 Web Computation Delays

Computation delays are significant. To quantify the computation delays in page loads, we replayed each page locally, without any network emulation, i.e., all object fetches took ≈ 0 ms. As shown in Figure 1, even without network delays, popular pages in developed and emerging markets have median load times of 2.7 and 3.8 seconds, respectively. Worse, 48% and 63% of pages require more than the 3 second load

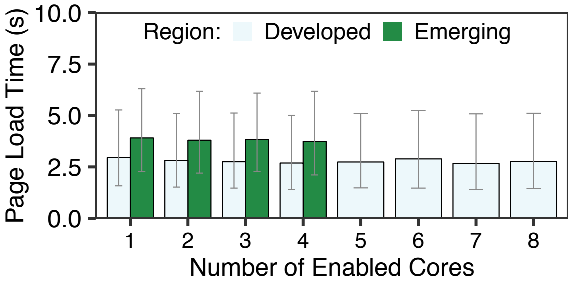


Figure 3: **Additional CPU cores have minimal impact on load times. The developed and emerging region phones have 8 and 4 cores. Bars list medians, with error bars for 25-75th percentiles.**

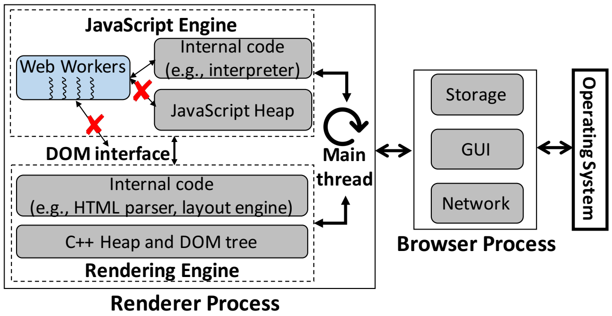


Figure 4: **Computation model for Chromium browsers.**

times that users tolerate [3]. These intolerable delays persist with metrics evaluating page visibility (i.e., Speed Index [40]), with 39% and 52% of pages in the developed and emerging regions taking more than 3 seconds to fully render.

JavaScript execution is the main culprit. To break down these high computation delays, we analyzed data from the browser's in-built profiler which lists the time spent performing various browser tasks including JavaScript execution, HTML parsing, rendering, and so on. Figure 2 illustrates our finding that JavaScript computation is the primary contributor, accounting for 52% and 58% of overall computation time for the median page in the two settings.

Browsers make poor use of CPU cores. Computation resources on mobile phones have globally increased in recent years, with improvements in both CPU clock speeds and total CPU cores. However, due to the energy constraints on mobile devices, increased core counts have been (and likely will continue to be) the primary source of improvements [77, 35]. For example, since their inception in 2016, Google's Pixel smartphones (our developed region phone) have improved clock speeds from 1.88 GHz to 2.15 GHz, while doubling the number of CPU cores (from 4 to 8). Similarly, the popular Redmi A series in India and Pakistan [4] (our emerging market phone) observed the same doubling in CPU cores (2 to 4) during that time period, while seeing only a modest clock speed improvement from 1.4 GHz to 1.75 GHz.

Unfortunately, although browsers can automatically benefit from clock speed improvements, we find that they fail to leverage available cores. To illustrate this, we iteratively disabled CPU cores on the phones in each setting and observed the impact on page load times. As shown in Figure 3, additional CPU cores yield minimal load time improvements, e.g., going from 1 to 8 cores on the Pixel 3 resulted in only a 8% speedup for the median page.

2.2 Browser Computation Model

To determine the origin of these computation inefficiencies, we must consider the computation model that browsers use today. Our discussion will be based on the Chromium framework [38], which powers the Chrome, Brave, Opera, and Edge browsers that account for 70% of the global market share [8, 5]. Figure 4 depicts Chromium's multi-process architecture. We focus on the *renderer process* which houses the Rendering and JavaScript engines, and thus embeds the core functionality for parsing and rendering pages.

The Rendering engine parses HTML code, issues fetches for referenced files (e.g., CSS, JavaScript, images), applies CSS styles, and renders content to the screen. During the HTML parse, the rendering engine builds a native representation of the HTML tree called the *DOM tree*, which contains a node per HTML tag. As the DOM tree is updated, the rendering engine recomputes a layout tree specifying onscreen positions for page content, and issues the corresponding paint updates to the browser process.

The JavaScript engine is responsible for parsing and interpreting JavaScript code specified in HTML `<script>` tags, either as inline code or referenced external files. During the page load, the JavaScript engine maintains a managed heap which stores both custom, page-defined JavaScript state and native JavaScript objects (e.g., `Dates` and `RegExps`). JavaScript code can initiate network fetches via the browser process (e.g., using `XMLHttpRequests`), and can also access the rendering engine's DOM tree (to update the UI) using the DOM interface. The DOM interface provides native methods for adding/removing nodes and altering node attributes; DOM nodes accessed via these methods are represented as native objects on the JavaScript heap.

The problem: single-threaded execution. JavaScript execution is single-threaded and non-preemptive [65, 64]. Worse, within a renderer process, all tasks across the two engines are coordinated to run on a single thread, called the *main thread*.[1] This lack of parallelism largely explains the poor use of CPU cores in §2.1. A primary reason for this suboptimal computation model is that the JavaScript language and DOM data structure (shared between the two engines) lack synchronization mechanisms (e.g., locks) to enable safe concurrency. Adding thread safety is feasible, but browsers have continually opted for a serial-access model to simplify page development. Browsers do create a separate renderer process per cross-domain `iframe` in a page (as per

[1] Some Chromium implementations move final-stage rendering tasks to raster/composite threads that create bitmaps of tiles to paint to the screen.

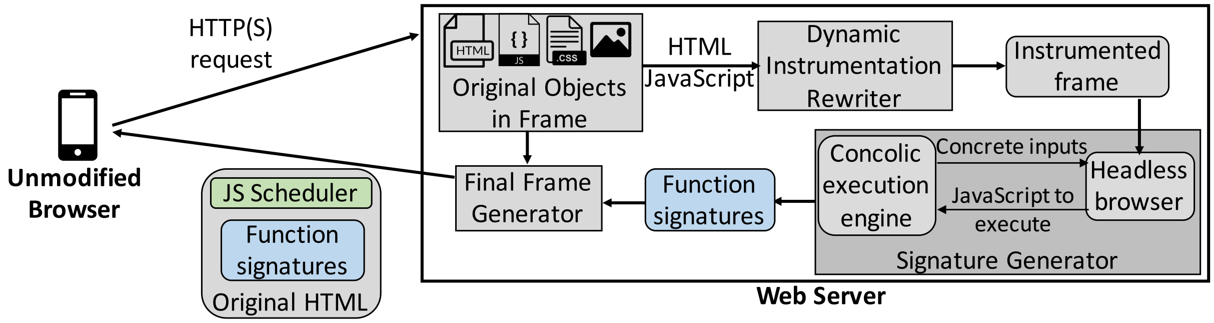


Figure 5: **Overview of the generation and fetch of each frame's top-level HTML file with Horcrux. Offline, servers collect a frame's files (both local and third-party), generate comprehensive state access signatures for each JavaScript function, and embed that information in their pages. Parallelism decisions are made online by unmodified client browsers using the Horcrux JavaScript scheduler, which manages computation offloads to Web Workers and maintains the page's JavaScript heap and DOM tree.**

# of Cores	% Speedup in Total JavaScript Runtime
2 cores	54%
4 cores	79%
8 cores	88%

Table 1: **Potential parallelism speedups with varying numbers of cores. Results list median speedups in total time to run all JavaScript computations per page in the developed region.**

the Same-origin content sharing policy [7]). However, for the median page in the Alexa top 10,000, the top-level frame accounts for 100% of JavaScript execution delays.

In summary, despite benefits regarding simplified page development, the single-threaded execution model that browsers impose results in significant underutilization of mobile device CPU cores, inflated computation delays, and degraded page load times. We expect this negative interaction to persist (and worsen) moving forward given the steady and unrelenting increase in the number of JavaScript bytes included in mobile web pages, e.g., there has been a 680% increase over the last 10 years [44].

3 Overview

Given the results in §2, a natural solution to alleviate client-side computation delays in mobile page loads is to *parallelize* JavaScript execution across a device's available CPU cores. However, not all workloads are amenable to parallel execution. In particular, we face the restriction that any introduced parallelism should preserve the page load behavior (and the final page state) that developers expected when writing their legacy pages—we call this property *safety*.

3.1 Potential Benefits

To estimate the potential benefits of parallelism with legacy pages, we analyzed the JavaScript code for each page in our corpus; in this section, we focus on page loads representative of those in developed regions, and we show later in §6 that similar benefits are achievable for page loads in emerging markets. Since JavaScript functions account for 94% of the JavaScript source code on the median page, our analysis operates at the granularity of functions, i.e., when splitting computations on a page across CPU cores, all code within a function runs sequentially on the same core. For completeness, we convert all code outside of functions into anonymous functions. For each function, we recorded both its runtime in a single load, as well as all accesses that it made to page state (in the JavaScript heap or DOM tree, as described below) in that load; §4.1.1 details the data collection process.

Using these logs, we estimated an *upper bound* on parallelism benefits by maximally packing function invocations to available cores and recording the resulting end-to-end computation times. To ensure safety (defined above), our analysis respects two constraints: 1) functions can run in parallel if they access disjoint subsets of page state or only read the same state, and 2) functions that exhibit state dependencies (i.e., access the same state and at least one writes to that state) execute in an order matching that in the legacy page load.

As shown in Table 1, legacy pages are highly amenable to safely reaping parallelism speedups. For example, distributing computation across 4 cores could bring a 75% reduction in the total time required to complete all JavaScript computations on the median page. These resources are now common in both developed and emerging markets [24].

3.2 Goals and Approach

To realize these benefits in practice, we seek an approach that minimizes the bar to adoption. As a result, requiring developers to rewrite pages [53] is a non-starter, given the complexities involved in manually reasoning about the impact of concurrently running portions of the JavaScript code on a page. Moreover, approaches that only require changes to browsers would either have to speculatively parallelize code [21, 56] or perform client-side analysis of JavaScript code (akin to the analysis that informed our estimated benefits above). As we show in §6.3, the overheads imposed by either strategy make them untenable, especially on energy-constrained phones.

Therefore, we pursue an approach which can safely parallelize JavaScript execution on *legacy* web pages with *unmodified* browsers. As shown in Figure 5, our solution, Hor-

Delay type	0.5 KB	1 KB	100 KB	1 MB
Startup	128 ms	155 ms	237 ms	317 ms
Value I/O	0 ms	1 ms	1 ms	7 ms

Table 2: **Web Worker overheads for different sizes of state transfers, i.e., source code size for startup delays and JavaScript object size for I/O delays.**

crux, only necessitates server-side changes that do not require developer participation to rewrite pages. Web servers perform the *expensive* task of tracking the state accessed (in the JavaScript heap or DOM tree) by every JavaScript function in a page frame, and include this information in that frame in the form of per-function *signatures*. Servers also embed a JavaScript (JS) scheduler library in the frames they serve, which enables unmodified client browsers to perform the *cheap* task of managing parallelism using function signatures obtained from servers. Dynamically determining the parallelization schedule at the client helps Horcrux account for information only available at runtime, e.g., the number of available threads and the control flows in the current load.

3.3 Challenges

The key building block in Horcrux is the widespread support in browsers for the Web Workers API [55], which allows the JavaScript engine to employ additional computation threads (Figure 4), as specified by a page's source code. Using Web Workers to parallelize JavaScript execution, however, presents numerous challenges that complicate achieving the idealistic parallelism benefits from above.

1. **Ensuring correctness.** Determining what JavaScript code can be *safely* offloaded requires a comprehensive understanding of how that code will access JavaScript heap and DOM state *in the current page load*. This, in turn, depends on the traversed control flows, which can vary due to both client-side (e.g., `Math.Random`) and server-side (e.g., cookies) sources of nondeterminism in JavaScript execution [59]. Missed state accesses can lead to dependency violations and broken pages.
2. **Constrained API.** Web Workers impose restricted computation models in two ways. First, due to the lack of synchronization mechanisms in JavaScript, Workers cannot access the JavaScript heap, and instead can only operate on values explicitly passed in by the browser's main thread (via `postMessages`). Thus, offloading computations to a Worker requires knowledge of precisely what state is required for those computations. Workers must then communicate computation results back to the main thread, which applies the results to the heap. Second, regardless of the state passed in, Workers cannot perform any DOM computations, including invocations of native DOM methods or operations on live DOM nodes referenced in the heap. We note that Workers can spawn and manage other Workers, but still must rely on the main thread for access to any global state.
3. **Offloading costs.** Lastly, operating Web Workers is not free. Instead, as shown in Table 2, spinning up a Web Worker can take over 100 ms, even for small amounts of source code being passed in. Pass-by-value I/O adds an additional several milliseconds, depending on the size of the transferred state. Moreover, JavaScript execution is non-preemptive; so, Workers that finish their tasks may go idle for long durations if the thread responsible for assigning them more tasks is busy.

We posit that, it is for these reasons that only a handful of the Alexa top 1,000 pages use Workers, despite support by commodity browsers for over 8 years. We next describe how Horcrux overcomes these issues to automatically parallelize JavaScript execution for legacy pages.

4 DESIGN

In designing Horcrux, we primarily need to answer two questions: 1) how to determine which JavaScript functions on a page can be executed in parallel without compromising correctness?, and 2) how to realize parallel execution at low overhead despite constraints placed by the browser's API? We present our solutions to these issues by separately describing server-side and client-side operation in Horcrux. Table 3 summarizes the main techniques underlying our design.

4.1 Server-side Operation

The goal of Horcrux's server-side component is to annotate page frames with per-function signatures that list the state that each function *might* access. Operating at a frame level, rather than at the granularity of entire pages, is in accordance with the browser's content sharing model [7, 64]. As in prior web optimizations that involve page alterations [63, 64, 54], Horcrux assumes access to a frame's source files. These files can be quickly collected either using a headless browser[2] or via integration into content management systems (for local files) [27, 89]. Source file collection and signature generation is retriggered based on hooks that many content management systems fire any time a (local) file-altering change is pushed [27, 89, 47, 54], e.g., for A/B testing; we discuss third-party content changes and personalization in §4.2.2.

4.1.1 Generating signatures

Since web servers cannot precisely predict the control flows that will arise in any particular page load (e.g., due to client-side nondeterminism), each function's signature must conservatively list *all possible* state accesses for that function. For this reason, we cannot directly apply recent web dependency tracking tools [63, 64] that rely purely on dynamic analysis to track data flows *in a given page load*. At the same time, pure static analysis approaches are ill-suited for JavaScript's dynamic typing, use of blackbox browser APIs,

[2] A headless browser performs all of the tasks that a normal browser performs during a page load except those that involve a GUI.

Goal	Techniques	Section
Ensure correctness	For each function, use concolic execution to identify union of the state it accesses across all control flows	§4.1.1
	Adapt offloading schedule during a page load to account for control flow in current load	§4.2.2
Account for API restrictions	Main browser thread centrally manages global page state, coordinates offloads, and performs unoffloadable (DOM) computations	§4.2.1, §4.2.3
	Intercept any Web Worker's DOM tree accesses and relay to the main thread	§4.2.3
	Use function signatures to determine what heap values to pass-by-value from main thread to workers and back	§4.2.3
Minimize overheads	Offline server-side generation of per-function signatures	§4.1.1
	Offload at the granularity of root functions	§4.1.2
	Dynamically determine offloading schedule based on function runtimes in current page load	§4.2.1

Table 3: **Overview of the main insights that Horcrux uses to address the challenges outlined in §3.**

and event-driven/asynchronous execution [75, 52]. For example, static analysis of variable name resolution is complicated by JavaScript statements that push objects to the front of the scope resolution chain (`with(obj){}`), or dynamically generate code (`eval()`). Similar issues arise from JavaScript's extensive use of variable aliasing for DOM objects, and the fact that property names are routinely accessed via dynamically-generated strings instead of static ones.

Thus, we turn to concolic execution [36, 80, 48], a variant of symbolic execution that executes programs concretely (rather than symbolically) while ensuring complete coverage of all control paths. A concolic execution engine loads a program with a concrete set of input values and observes its execution; §5 describes the inputs we consider, including browser state (e.g., cookies, screen size) and nondeterministic functions. Each input value and program-generated value is also given a symbolic expression constrained only by its type. For example, an input integer `a` may get a concrete value of 10 and an expression of $0 \leq \texttt{a} \leq 2^{32} - 1$; symbolic expressions for a given variable are inherited by others via assignment statements. At each branch condition, the execution follows the appropriate path based on the current program state. In addition, the engine restricts the symbolic expressions for the values that influenced the chosen path according to the branching predicate. Once the program completes, the engine performs a backwards scan through the executed code, selects branching decisions to invert, and inverts the relevant symbolic expressions; an SMT solver [26] then generates concrete input values that satisfy the new constraints. This process repeats until all paths are explored.

Note that, for efficiency, many recent symbolic execution tools opt for a form of concolic execution, rather than a purely symobolic approach [20]. More specifically, concolic execution engines consult the expensive constraint solver only at the end of each path (rather than at intermediate branches), and eliminate the need to accurately model each input source to a program (and the ensuing traversal of paths that arise due to modeling errors).

To generate function signatures, in addition to the default output of a concolic execution engine – a list of potential control paths, with a concrete set of inputs to force each one – we must also log the state accessed by each path. To do this, prior to concolic execution, Horcrux instruments the JavaScript source code to log all accesses to state in both the JavaScript heap and DOM tree; our instrumentation matches recent dynamic analysis tools [63, 66, 64], but with the following differences based on our parallelism use case.

- First, we care not just about the state that remains at the end of the page load [64], but also any state accessible by multiple functions during a page load. Hence, in addition to global heap objects, Horcrux tracks all accesses to closure state: non-global state that is defined by a function X and is accessible by all nested functions that execute in X's enclosed scope (anytime during the page load) [57].
- Since signatures will ultimately be used for pass-by-value offloading to Workers, only the finest granularity of accesses are logged. For instance, if object `a`'s "foo" property is read, Horcrux would log a read to `a.foo`, not `a`.
- For the DOM tree, Horcrux adopts a coarser approach than prior work. Instead of logging reads and writes to individual nodes in the DOM tree, Horcrux only logs whether a function accesses any live DOM nodes, either via DOM methods or references on the heap, and if so, whether they are reads or writes. Tracking at the coarse granularity of accesses to the entire DOM tree is conservative with respect to parallelism. However, finer-grained tracking is not beneficial because, as we explain later, our design has the browser's main thread serialize all DOM operations.

4.1.2 Signature granularity

Ideally, to limit client-side bookkeeping overheads, signatures should match the granularity at which computation is offloaded. However, determining the appropriate offloading granularity is challenging. On the one hand, fine-grained offloading reduces the chance that offloadable computations access shared state, thereby improving the potential parallelism and use of available Workers. On the other hand, finer granularities imply increased coordination overheads.

To address this tradeoff, Horcrux generates signatures (and offloads computation) at the granularity of *root function* invocations, i.e., invocations made directly from the global JavaScript scope. The signature for each root function invocation includes the state accessed not only by that function, but also by any nested functions that are invoked in the call chain until the global scope is reached again.

Root function signatures are desirable for two reasons. First, they leverage our finding that functions already account for 94% of JavaScript code on the median page (§3) and thus provide a natural granularity for offloading; as in §3, Horcrux servers wrap all JavaScript code outside of any function into anonymous functions. Second, and more importantly, root functions impose far smaller offloading overheads compared to finer-grained function-level offloading, while enabling comparable parallelism benefits: the number of offloads drops by 4×, while the median potential benefits remain within 27% of those in Table 1. The reason is that there often exists significant state sharing within the invocations for a given root function (and its nested components), but less so across root functions, enabling parallelism.

4.2 Client-side Operation

Even with function signatures, a server cannot precompute a parallel execution schedule because the precise control flow, and hence, the set of functions executed and their runtimes, will vary across loads. Instead, Horcrux employs a client-side JavaScript computation scheduling library that unmodified browsers can run to dynamically make parallelism decisions based on signatures and the aforementioned runtime information. The key challenges are in efficiently ensuring correctness while offloading to multiple Workers, and handling the fact that signatures may be missing for certain functions. We next discuss how Horcrux addresses these challenges.

4.2.1 Dynamic scheduling

To load a page frame, any unmodified browser first downloads the top-level HTML, whose initial tag is an inline `<script>` housing Horcrux's scheduler library and all root function signatures. The scheduler runs on the browser's main thread and begins by asynchronously creating a pool of uninitialized Workers. This helps hide the primary overhead of spawning Workers amongst unavoidable delays for parsing initial HTML tags and fetching files they reference.

The scheduler then operates entirely in event-driven mode, whereby it waits for incoming postmessages specifying computations to perform or those that have completed, and makes subsequent offloading decisions. Importantly, to keep the main thread as idle as possible, the scheduler offloads *all* computations that Web Workers can support, and is primarily responsible for managing Workers and maintaining the page's global JavaScript heap and DOM state. This helps adapt the parallelization schedule within any page load to the runtimes of every root function in that load. The reason is that the main thread will be available to quickly assign a new function invocation (if one exists) to any worker that completes executing the function previously assigned to it.

Once the scheduler is defined, the browser operates normally, recursively fetching and evaluating referenced HTML, JavaScript, CSS, and image files. However, all JavaScript function invocations are modified to pass through the scheduler for offloading. More specifically, each root function is rewritten such that, upon invocation, the function sends a post message to the scheduler specifying its original source code and that of any nested functions. Special care is taken for asynchronous functions (e.g., timers) whose invocations are regulated by the browser's internal event queue which the scheduler does not have access to. To ensure visibility to such functions, the downloaded page includes shims around registration mechanisms for asynchronous functions, e.g., `setTimeout()`. Each shim modifies registered functions to send messages to the scheduler upon invocation.

Each time a root function is invoked, the scheduler uses its signature to determine whether or not it can be immediately offloaded. If not, the function is stored in an in-memory queue of *ordered*, to-be-invoked functions along with its signature. Functions are not offloadable if there are no available Workers, or if they might access state that is being modified by an already-offloaded or queued function. Note that functions that may access the DOM can be offloaded in parallel; we will discuss how to ensure safety in these cases shortly.

Regardless of the decision for a given invocation, the browser continues its execution. At first glance, it may appear that continuation after a queued invocation may generate errors since the queued function could alter the set of downstream invocations. However, recall that Horcrux offloads at the granularity of root functions—any nested invocations are already offloaded, and the ordering of root functions is mostly predefined by the page's source code. There are two exceptions. First, a function can alter downstream source code using `document.write()`; to handle this, the scheduler synchronously offloads such functions, thereby blocking downstream execution. Second, a root function can register an asynchronous function with a 0-ms timer—such functions are intended to run immediately after the current invocation. For this, the root signature includes state accesses for the 0-ms timeout functions they define. Once the root function is discovered, the scheduler adds a placeholder for the timeout function to its queue, thereby blocking downstream invocations that share state with the timeout function.

4.2.2 Handling Missing Signatures

We have assumed so far that the HTML file of every page frame includes accurate signatures for all JavaScript functions executed in that frame. This may not always hold.

- *Stale signatures.* A frame can include JavaScript content from multiple origins, and to preserve HTTPS content integrity and security [67], Horcrux has each origin serve its own files directly to clients. A third-party origin may update a script without explicitly informing the top-level origin to regenerate signatures. We expect this to be rare for two reasons. First, JavaScript files often have long cache lives (median of 1 day in our corpus), indicating infrequent changes. Second, scripts in a frame can share state [7].

Thus, even today, if a third-party origin significantly alters a script it serves, this should be communicated to other origins to avoid unexpected or broken behavior.

- *Dynamically-generated or personalized scripts.* JavaScript files may be created or personalized in response to user requests [54], e.g., based on Cookies. Unfortunately, generating signatures during client page loads would be far too slow. To handle this, for dynamically-generated or personalized first-party content, Horcrux could perform concolic execution on server-side content generation logic to determine the execution paths for all variants of a given response (§5). Third-party content of this type may result in missing signatures since the top-level origin does not have access to a user's third-party Cookies (or personalized content). However, many browsers preclude third-party Cookies in frames to prevent the tracking of users across sites [23].
- *Timeouts of concolic execution.* Given infinite time, concolic execution is guaranteed to explore all possible JavaScript execution paths in a page [36, 80]. However, the process may timeout, either during the execution of a given path (if the SMT solver cannot invert a branch condition), or less likely, due to a time bound placed on overall signature generation. Regardless of the reason, the effect is a potentially missing or incomplete signature. Such timeouts did not arise (i.e., concolic execution completed) for any pages in our experiments (§6.1). However, in the event that concolic execution does not complete, Horcrux could detect such timeouts *prior* to serving content to clients, and could thus address the corresponding missing signatures (described below) or revert to a normal page load.

Horcrux accounts for missing or inaccurate signatures in two ways. First, any underexplored function *X* is assigned a signature of $\star$, indicating that *X* may access all page state. This overconstrains the client's load, but ensures correctness: the client will execute *X* serially, and will also serialize downstream functions since *X* might alter the state they access. Second, signatures are keyed by a hash of the function's source code. Invocations without matching signatures are assigned signatures of $\star$ by Horcrux's client-side scheduler.

4.2.3 Function Offloading and Execution

Lastly, we discuss the mechanics of how every function invocation that is offloaded by the Horcrux scheduler is executed in a Web Worker. Figure 6 illustrates this process.

For each offloadable function, the scheduler uses its signature to generate a JSON package listing the information that the Worker will require for execution, i.e., the source code (including nested functions) and the current values for the function's read state. The source code is modified such that, upon completion, values in the write state are gathered and sent to the main thread (as execution results). In addition, closure values in the read state are embedded into the corresponding function's source code. Upon reception, the post message handler inside the Worker sets up the read state in the Worker's global scope via assignment statements, and runs the code using the browser's `Function` constructor.

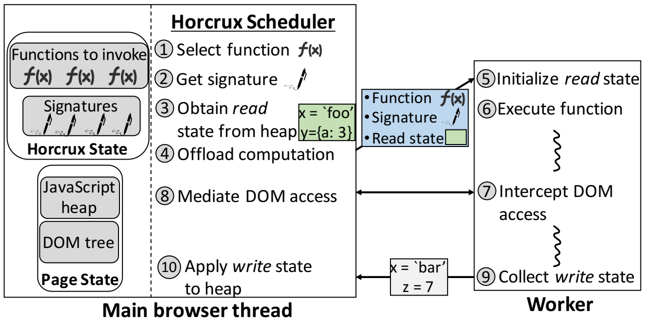


Figure 6: **A single function offload with Horcrux.**

For the most part, functions execute normally, with JavaScript heap accesses hitting in the Worker's global scope, or for nested functions, in the scope of their parents–recall that Horcrux offloads entire root functions so nesting relationships are preserved. However, the key difference is with respect to DOM accesses: workers cannot call native DOM methods or operate on live DOM nodes referenced in the heap (§3.2). To handle this, all DOM computations are mediated by the scheduler and are serially applied to the live DOM tree. To intercept DOM computations, Horcrux includes shims around all DOM methods in each Worker environment; returned DOM nodes are replaced with proxy objects to interpose on direct accesses. Each intercepted DOM access blocks execution in the Worker, and is sent to the scheduler where it is queued; blocking is enforced using JavaScript generator functions [58].

The scheduler grants readers-writer locks to each Worker that may need to access the DOM tree (as per their signatures). Locks are granted in the order that the scheduler receives function invocations; note that this may not match the order in which functions are offloaded, but it preserves the relative ordering of DOM updates seen in a normal page load. As a concrete example, consider a function `a` that reads from the DOM, and a later function `b` that writes to the DOM. `b` may attempt to access the DOM first (e.g., if it is offloaded earlier or its DOM access occurs early in the function), but the scheduler will block it and wait to grant the lock to `a` first. Workers release DOM locks at the end of their execution. In essence, root functions that only read from the DOM tree can run in parallel, although their constituent DOM accesses are serialized on the browser's main thread. Root functions that write to the DOM are run serially with respect to other DOM-accessing root functions (to match the relative ordering of DOM updates in an unmodified load); for context, only 7.4% of the root functions on the median page in our corpus involve DOM writes, mitigating the effects of such serialization. Importantly, locking is done at the granularity of entire root invocations because the scheduler does not definitively know whether a given function will access the DOM in the current load (and if so, how many times); signatures only list that a DOM access *may* occur.

Once a function completes execution, the Web Worker sends its computation results (i.e., its write state) to the scheduler. The Worker then clears any state in its scope in order to be ready for the next offload. Upon receiving computation results, the scheduler applies the writes to the global JavaScript heap; recall that DOM writes have already been made. One subtlety here is with respect to scope, and closure state in particular. The scheduler can access and apply computation results to the global scope's heap. However, root functions can also modify shared closure variables which are not accessible from the global scope (§4.1). For such writes, the scheduler maintains a global hash map listing the latest closure values. This map is updated as Workers complete computations, and is also queried to obtain read values when offloading; note that correctness is ensured because all offloads and computation results pass through the scheduler.

Finally, once the scheduler applies computation results to the JavaScript heap, it scans through its queue of ordered, to-be-executed functions and offloads the next one that can safely run. Given the serialization of DOM computations described above, if a function that writes to the DOM is queued, the scheduler prioritizes queued functions that do not access the DOM (and thus won't incur locking delays).

4.3 Discussion

Key to Horcrux's operation is the decision to maximally offload and parallelize JavaScript computations at the granularity of root functions. Recall that this decision was motivated by our analysis of the JavaScript computation on the pages in our experimental corpus (§3.1 and §4.2), which revealed that root function offloading favorably balances client-side overheads (e.g., pass-by-value I/O, main thread responsiveness) with the achievable speedups from parallelization.

However, these decisions may not deliver the largest speedups for certain pages. For example, root function-level offloading might be too restrictive and forego significant parallelism benefits, e.g., if a root function includes two nested functions that access entirely disjoint state but both involve significant runtime. Similarly, the root functions for certain pages could each embed only a single nested function, thereby inflating offloading costs relative to parallelization speedups, and potentially harming overall performance.

Although we did not observe these behaviors for any of the pages in our experiments (§6), we note that developers could perform analyses similar to the one presented in §3.1 to determine whether automatic parallelization of JavaScript code is desirable for (i.e., can speed up) their pages, and if so, what the best offloading granularity is. Importantly, these analyses do not require further instrumentation of web pages, and instead can directly leverage Horcrux's signatures, the per-function runtimes reported by in-built browser profilers, and the relatively stable offloading costs reported in Table 2.

5 Implementation

Horcrux instruments JavaScript source code to generate signatures and prepare frames using Beautiful Soup [78], Esprima [43], and Estraverse [84]. To employ concolic execution, we use a modified version of Oblique [48], which runs atop a headless version of Google Chrome (v85) and the ExpoSE JavaScript concolic execution engine [50]. Our Oblique implementation considers inputs specified by HTTP headers (e.g., Cookie, User-Agent, Origin, Host), the device (e.g., screen coordinates), and built-in browser APIs including nondeterministic functions [59] (e.g., `Math.Random`) and DOM methods. Input values suggested by the SMT solver are fed into the page load via either 1) rewritten HTTP headers, or 2) shims for browser APIs.

We grant Oblique a maximum of 10 mins to consider a given execution path, and 45 mins to explore all paths for a given page; we find that these time values are sufficient to ensure that concolic execution completes for all of the pages in our experimental corpora (§6.1). Signatures from each load are sent to a dedicated analysis server for aggregation. Since our current implementation operates directly on downloaded page source code and not live web backends (§6.1), Horcrux eschews Oblique's ability to perform concolic execution on server-side application logic. In total, Horcrux's implementation involves 5.6k LOC in addition to Oblique, including 4.5k for dynamic tracing (both static instrumentation and runtime tracking) and 1.1k for client-side scheduling.

Overheads. On the client-side, Horcrux inflates page sizes by 13 KB at the median (when using Brotli compression [41]). The scheduler accounts for 3 KB of that.

6 Evaluation

We empirically evaluate Horcrux across a wide range of real pages, live mobile networks, and phones from both developed and emerging markets. Our key findings are:

- Horcrux reduces median browser computation delays by 31-44% (0.9-1.5 secs), which translates to page load time and Speed Index improvements of 18-29% and 24-37%. Improvements grow with warm browser caches (§6.2).
- Horcrux delivers larger speedups than prior web optimizations that 1) reduce required computations (by 1.7-2.1×), 2) speculatively parallelize computations (by 1.3-1.6×), and 3) mask network round trips (by 1.4-2.1×); Horcrux is complementary to network optimizations and running them together lowers load times by 31-45% (§6.3).
- Although the median page has 12 possible execution paths, Horcrux's reliance on conservative signatures (for correctness) only foregoes 7-10% of speedups compared to using signatures that target a specific load (§6.4).
- Horcrux is highly amenable to partial deployment: benefits are within 2% of total adoption when only a page's top-level origin runs Horcrux. Benefits persist for personalized pages and desktop settings (§6.5).

6.1 Methodology

We evaluated Horcrux in two different scenarios:

- **Developed regions.** We consider 700 US pages, randomly selected from and equally distributed amongst the following sources: popular landing pages from the Alexa [12] and Tranco [49] top 1000 lists, popular interior pages from the Hispar 100,000 list [16], and less popular pages (landing and interior) from the 0.5 million-site DMOZ directory [1]. Thus, our corpus involves diversity in terms of both page popularity and location within a website (i.e., landing vs. interior). From this set, we report results for the 582 pages that our current implementation could generate accurate signatures for. More specifically, we removed pages for the following reasons: 1) inaccurate signatures due to unsupported language features, which led to premature JavaScript termination (92) or rendering defects (22), and 2) unsupported features with Oblique (4). For all of the remaining pages, Horcrux's concolic execution and overall signature generation completes, the total JavaScript runtime with Horcrux falls within a standard deviation of that in default loads, and the final rendered page is unchanged. Our experiments consider two powerful phones, a Pixel 3 (Android Pie; 2.0 GHz octa-core processor; 4 GB RAM) and a Galaxy Note 8 (Android Oreo; 2.4 GHz octa-core; 6 GB RAM). For space, we only present results for the Pixel 3, but note comparable results with the Galaxy Note 8.
- **Emerging regions.** Web experiences in emerging regions often comprise different page compositions and devices than those considered above [24, 10, 11, 90]. To mimic such scenarios, we focus on a single emerging region: Pakistan. We consider a corpus of 100 landing and interior pages (50 each) selected from the Alexa Top 500 sites in Pakistan. Our evaluation uses the Redmi 6A phone (Android Oreo; 2.0 GHz quad-core processor; 2 GB RAM) that is popular in the region [4]. As per the same correctness checks as above, we report numbers on 91 pages.

Unless otherwise noted, page loads were run with Google Chrome for Android (v85). Mobile-optimized (including AMP [37]) pages were always used when available.

To create a reproducible test environment, and because Horcrux involves page rewriting, we use the Mahimahi web record-and-replay tool [68]. Emerging regions pages were recorded using a VPN to mimic a client in Pakistan. As described in §4, Horcrux's signature generation and page rewriting were performed offline. To replay pages, we hosted the Mahimahi replay environment on a desktop machine. Our phones were connected to the desktop via USB tethering and live Verizon LTE and WiFi networks with strong signal strength; LTE speeds for emerging regions experiments were throttled to Pakistan's 7 Mbps average [70]. We used Lighthouse [42] to initiate page loads via the USB connection, and all page load traffic traversed the wireless networks.

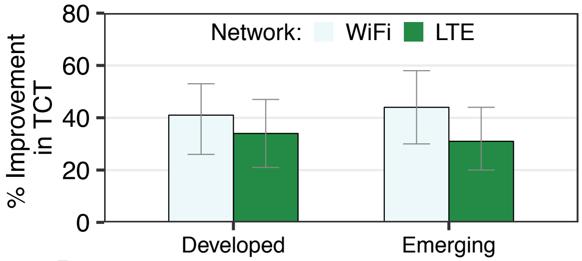
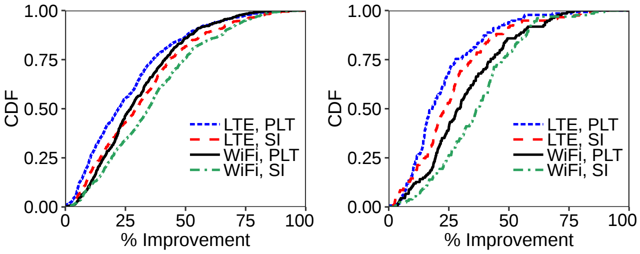


Figure 7: **Cold cache TCT improvements over default page loads. Bars list medians, with error bars for 25-75th percentiles.**



(a) **Developed region.** (b) **Emerging region.**

Figure 8: **Distributions of cold cache per-page improvements with Horcrux vs. default page loads.**

We evaluated Horcrux on multiple web performance metrics: 1) *Total Computation Time* (TCT), or the critical path of time spent parsing/executing source files and rendering the page, 2) *Page Load Time* (PLT) measured as the time between the `navigationStart` and `onload` JavaScript events, and 3) *Speed Index* (SI) [40] which captures the time required to progressively render the pixels in the initial viewport to their final form. TCT and PLT are measured using the browser profiler, while SI was reported by Lighthouse. In all experiments, we load each page three times with each system under test, rotating amongst them in a round robin; we report numbers per system from the load with the median TCT.

6.2 Page Load Speedups

Cold cache. Figure 7 illustrates Horcrux's ability to reduce browser computation delays compared to default page loads. TCT reductions were 41% (1.0 sec) and 44% (1.5 sec) for the median page in the developed and emerging region's WiFi settings, respectively. Improvements were 34% and 31% with LTE. Figure 8 shows how these computation speedups translate into faster end-to-end (i.e., including network delays) loads. For example, on WiFi, median improvements in the developed region setting were 27% for PLT and 35% for SI. Despite the lower CPU clock speeds, these numbers only marginally increase to 29% and 37% in the emerging region. Further improvements were hindered primarily by the lower number of available cores (and thus ability to parallelize). Benefits with Horcrux on LTE were comparable, but consistently lower than with WiFi. For example, in the developed region, PLT and SI speedups were 22% and 29%. The reason is that network delays (which Horcrux does not improve) account for larger fractions of end-to-end load times on LTE.

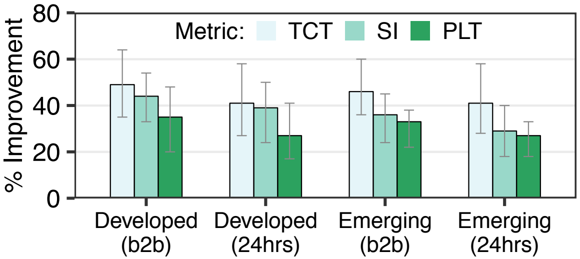


Figure 9: **Warm cache speedups over default page loads on LTE. Bars list medians, with error bars for 25-75th percentiles.**

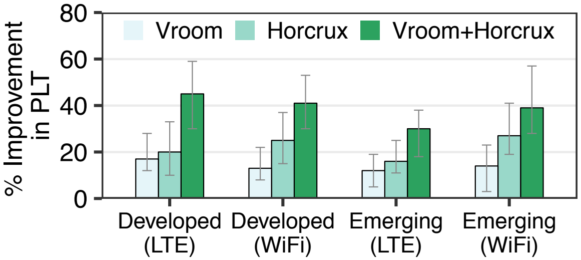


Figure 10: **Horcrux vs. Vroom [79] over LTE networks. Bars list medians, with error bars for 25-75th percentiles.**

Warm cache. Figure 9 shows Horcrux's speedups in different browser caching scenarios. We consider back-to-back page loads, as well as those separated by 12 and 24 hours. As shown, Horcrux's improvements grow as browsers house more objects in their caches. For example, in the back-to-back scenario, PLT and SI improvements in the developed region's LTE setting were 35% and 44%; for context, these improvements were 22% and 29% with cold caches. Improvements drop to 27% and 39% in the 24-hour warm cache scenario. The reason is that more cache hits lead to lower network delays and computation dominating end-to-end performance. In addition, cache hits enable browsers to begin processing JavaScript files earlier. This, in turn, provides Horcrux's scheduler with more invocation options at any time, thereby increasing the amount of potential parallelism.

6.3 Comparison to State-of-the-Art

Network optimizations. We first considered Vroom [79], a system in which web servers intelligently use HTTP/2's server push and preload features to aid clients in discovering (and downloading) required files ahead of time. Thus, Vroom is primarily a network-focused optimization. However, key to Vroom's benefits is the improved CPU utilization that results from eliminating blocking network fetches.

As shown in Figure 10, Horcrux delivers larger speedups than Vroom. For example, in the developed region, median PLT speedups with Horcrux are 2.1× and 1.3× higher than Vroom's on WiFi and LTE, respectively. In the LTE setting, Vroom delivers larger PLT speedups for 9% of pages. The reason is that network delays play a larger role in end-to-end load times for these pages, either due to less computation or more required network fetches. This drops to 1% and 3% when we move to the developed region's WiFi network or the emerging market's LTE network; in both cases, compute becomes more of a determinant of overall delays. Importantly, Figure 10 also confirms that Horcrux and Vroom are largely complementary to one another, with the combined systems outperforming each in isolation.

System	**Developed**	**Emerging**
Horcrux	1.63 (1.98)	2.15 (2.37)
Prepack [32]	2.19 (2.47)	2.82 (3.36)
Speculative parallelization	2.01 (2.28)	2.50 (3.07)

Table 4: **Comparing Horcrux with prior compute optimizations. Results are for WiFi networks and list median (75th) percentile TCTs in seconds.**

Reducing required computations. Prepack [32] is a server-side system that reduces the amount of JavaScript computation that clients must perform to load pages. To do this, Prepack performs static analysis on a page's JavaScript code, identifies expressions whose results are statically computable, and replaces those expressions with equivalent but simpler versions that remove intermediate computations. Importantly, computations involving client-side or nondeterministic state are unmodified; this helps Prepack preserve page behavior and correctness, unlike other computation reduction systems (§7). As shown in Table 4, Horcrux is more effective at reducing computation delays than Prepack: median TCTs are 26% and 24% lower with Horcrux in the developed and emerging regions, respectively.

Speculative parallelism. Prior efforts to increase parallelism in page loads (§7) primarily rely on speculative decisions about what can run in parallel, and runtime checks to detect (and revert from) dependency violations. Although these systems do not target all JavaScript execution, we considered a baseline that employs a similar parallelism strategy for JavaScript computation. Our baseline opportunistically parallelizes all root function invocations, and uses JavaScript proxy objects to track state accesses in each Worker. Any parallelized computations that share state are discarded, and the corresponding functions are rerun serially on the main browser thread. As shown in Table 4, Horcrux delivers superior median TCT values that are 14-19% lower across the two regions. The reason is twofold. First, proxy-based tracking to ensure correctness adds 10% overhead to JavaScript runtimes. Second, any speculation errors result in serial execution on the main thread and wasted computation (and thus, more overall computation). Using the setup in §6.5, we observe that this wasted computation inflates mobile device energy consumption by 9% for the median page on WiFi.

6.4 Understanding Horcrux's Benefits

Dissecting Horcrux's speedups. We analyzed Horcrux's behavior (and improvements) along three different axes. We focus on the developed region, but note that the trends hold for the emerging region setting. First, as expected, Horcrux's improvements are larger for pages that require more computation to load. For instance, with WiFi, median PLT improvements with Horcrux were 35% for pages with more than 3 seconds of computation time, as compared to 23% for pages that did not meet that criteria. This divide carries over to different page types as well: improvements were 15% higher for interior pages than landing pages. The reason is that interior pages often involve more computation [16, 64].

Second, within each load, we investigated the degree of parallelism that Horcrux achieves for JavaScript computation. For the median page, when loaded over WiFi, Horcrux reaches a maximum of 6 concurrent Web Workers; this drops to 4 on LTE due to the aforementioned network delays limiting the scheduler's parallelism options.

Third, in addition to JavaScript parallelization, Horcrux reduces TCT by freeing the browser's main thread for rendering tasks. To understand the contribution of each source to Horcrux's speedups, we analyzed the browser's computation profiler. Overall, we find that both sources provide substantial benefits. For instance, on WiFi, Horcrux shrinks effective JavaScript computation times by a median of 42% (557 ms), and decreases end-to-end rendering delays by 36% (465 ms).

Server-side overheads. Signature generation took 33 minutes for the median page, and involved two primary overheads: the median page involved exploring 12 different execution paths via concolic execution, and our dynamic instrumentation (incurred in each load) inflated load times by 44%. These non-negligible delays are why Horcrux performs comprehensive signature generation offline, on servers. To understand how often servers have to incur these overheads, we recorded a random set of 50 pages from our emerging region's corpus every 12 hours for 1 week. The median page's signatures remained unchanged for the entire duration, in part due to Horcrux's coarse-grained DOM tracking which is unaffected by changes to HTML state (e.g., headlines).

Cost of conservative signatures. Horcrux relies on conservative signatures that list the state accesses across all possible control flows. While this ensures correctness, it may overconstrain a client load that traverses only a subset of those control flows. To understand the impact of this conservative strategy, we compared Horcrux with a variant that generates signatures for the precise control flows traversed in the target client load. Surprisingly, we observe that Horcrux's conservative behavior results in only mild performance degradations: improvements drop by 10% and 7% for PLT and SI, respectively, for the developed region WiFi setting. The reason is that conservative signatures typically either add only a few extra state accesses to a given root function, or many that are only accessed for short durations (i.e., within a root function)—neither significantly restricts parallelism.

6.5 Additional Results

Partial deployment. Our results thus far assumed that each frame in a page adopts Horcrux, i.e., embeds Horcrux's scheduler and signatures in the HTML. Figure 11 shows Horcrux's benefits when only the top-level origin for the page participates—this represents the simplest deployment scenario as the top-level origin is directly incentivized to accelerate loads of its pages. In this scenario, all JavaScript code in third-party-owned frames runs serially; JavaScript in the main frame can still be safely parallelized as browsers prevent cross-frame state sharing [7]. As shown, most of Horcrux's benefits persist, despite the lack of adoption by third-party frames. For example, in the developed region's WiFi setting, median SI benefits are within 2% of those with total adoption. The reason is that most JavaScript runtime (100% on the median page) resides in the page's main frame.

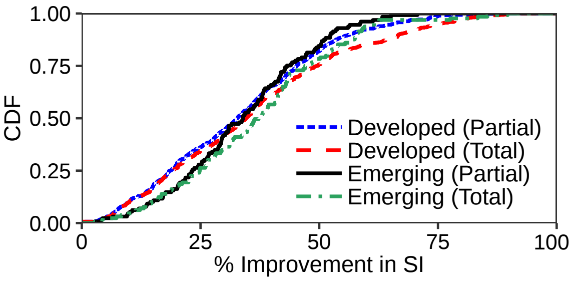


Figure 11: **Evaluating Horcrux when only a page's top-level origin participates. Results are for WiFi networks.**

Personalized pages. To evaluate Horcrux in settings where pages dynamically generate or personalize their content, we selected 20 pages from our developed region's corpus that supported user accounts. For each page, we created two user accounts, selecting different preferences when possible, e.g., order results based on time or popularity. For every file that does not appear in both loads, or whose content is different across the page versions, we assign its constituent functions signatures of $\star$ (§4.2.2). Overall, we observe that such personalization has minimal impact on Horcrux's speedups: in the WiFi setting, Horcrux's median load time benefits drop by only 4%. The reason is that only 6% of computation delays are accounted for by personalized scripts.

Energy savings. We connected our Pixel 3 phone to a Monsoon power monitor [60] and loaded the pages in our developed region corpus. With cold caches, Horcrux's speedups drop median per-page energy usage by 12% and 15% on WiFi and LTE. Savings are primarily from accelerating end-to-end computation (and load times), which results in lower active durations for WiFi or LTE radios.

Desktop page loads. Horcrux's acceleration techniques can also speed up desktop page loads. To evaluate this, we recorded desktop versions for the pages in our developed region corpus, and loaded them using a Dell G5 desktop and a wired network connection. We find that Horcrux reduces median TCT by 39% (0.52 secs). These speedups translate to PLT and SI improvements of 25% and 31%, respectively. At first glance, these improvements may appear surprising given the faster CPU clock speeds that desktops possess. However, desktops also possess more cores and load pages with more JavaScript computation [44], enabling more parallelism.

7 Related Work

Parallelization efforts. ParaScript [56] and others [61] leverage new runtimes and compiler information to speculatively parallelize iterations for hot loops in long-running JavaScript code (not page loads, where compilation overheads are too costly). In contrast, Horcrux operates with unmodified browsers, targets parallelism for general JavaScript code beyond loop iterations, and sidesteps the significant overheads of speculation errors and runtime checks (§6.3) by using conservative signatures. Zoomm [21] and Adrenaline [53] leave JavaScript execution unchanged, and instead parallelize tasks such as CSS rule parsing. These systems are orthogonal to Horcrux, which focuses entirely on JavaScript parallelization. Lastly, several libraries [6, 9] aid developers in writing parallel JavaScript code by abstracting inter-worker messaging. However, developers are responsible for identifying and enforcing (safe) parallelism decisions—Horcrux automates these tasks for legacy pages.

Reducing web computation overheads. Prior measurement studies have analyzed the performance of mobile web browsers [85, 62, 24, 73]. Like us (§2), they find that browser computations are a primary contributor to high page load times. In response to these studies, three separate lines of work have aimed to alleviate browser computation delays. First, certain sites have manually developed mobile-optimized versions of their pages using restricted forms of HTML, JavaScript, and CSS, e.g., according to the Google AMP standard [37, 46]. In contrast, Horcrux accelerates legacy pages without developer effort. Further, we find that Horcrux is able to accelerate the loading of AMP pages, which constitute 27% of our corpora.

Second, some systems [13, 87, 71, 22] offload computation tasks to well-provisioned proxy servers, which return computation *results* that are fast to apply. Though effective, such systems pose significant scalability challenges to support large numbers of mobile clients [82]. Worse, by relying on (often third-party) proxy servers, these systems violate HTTPS' end-to-end security guarantees [67]; clients must trust proxies to preserve the integrity of their HTTPS objects, and also must share private Cookies to accelerate personalized page content. In contrast, Horcrux is HTTPS-compliant.

Third, systems like Prophecy [64] enable servers to return post-processed page files that elide intermediate computations. However, content alterations with these systems may break page functionality [10], particularly for pages that adapt execution based on client-side state that servers are unaware of, e.g., localStorage. In contrast, Horcrux does not alter the set of computations required to load a page, and instead aims to execute those computations more efficiently.

Network optimizations for the web. Systems such as Alohamora [47], Vroom [79], and others [29, 86] leverage HTTP/2's server push and preload features to proactively serve files to clients in anticipation of future requests (thereby hiding download delays). Fawkes [54] develops static HTML templates that can be rendered while dynamic data is fetched. Polaris [63] and Klotski [19] reorder network requests to minimize the number of effective round trips while respecting inter-object dependencies. Cloud browsers [83, 67, 68] shift network round trips to wired proxy server links. Content delivery networks [69, 33] serve popular objects from proxy servers that are geographically close to clients, while compression proxies [10, 81, 72] selectively compress objects in-flight between servers and clients. Lastly, a handful of systems prefetch content according to predicted user browsing behavior [74, 51, 88]. As shown in §6.3, these efforts are complementary to Horcrux, which reduces browser computation delays by parallelizing JavaScript execution. Further, recall that computation delays often exceed user tolerance levels on their own (§2).

Concolic execution for web optimization. Like Horcrux, Oblique [48] uses concolic execution to accelerate web page loads. Indeed, Horcrux's server-side component builds atop Oblique's JavaScript concolic execution engine by adding dynamic instrumentation to capture per-function signatures (§5). However, despite this similarity, Oblique and Horcrux target different delays in the page load process: Oblique enables third-party servers to securely prefetch URLs that a client will need during a page load (hiding the associated network fetch delays), while Horcrux parallelizes the JavaScript execution required to load a page (reducing the associated computation delays). Consequently, as with other network-focused optimizations (§6.3), Oblique can run alongside Horcrux to provide complementary benefits.

8 Conclusion

Horcrux automatically parallelizes JavaScript computations in legacy pages to enable unmodified browsers to leverage the multiple CPU cores available on commodity phones. To account for the non-determinism in page loads and the constraints of the browser's API for parallelism, Horcrux employs a judicious split between clients and servers. Servers perform concolic execution of JavaScript code to conservatively identify parallelism opportunities based on potential state accesses, while clients use those insights along with runtime information to efficiently manage parallelism. Across browsing scenarios in developed and emerging regions, Horcrux reduced median browser computation delays and load times by 31-44% and 18-37%.

Acknowledgements: We thank James Mickens, Amit Levy, Anirudh Sivaraman, and Harry Xu for their valuable feedback on earlier drafts of the paper, as well as Blake Loring for useful discussions on the ExpoSE JavaScript concolic execution engine. We also thank our shepherd, Ryan Huang, and the anonymous OSDI reviewers for their constructive comments. This work was supported in part by NSF grants CNS-1943621 and CNS-2006437.

References

[1] Directory of the web. https://dmoz-odp.org/.

[2] Web Workers. https://w3c.github.io/workers/, 2017.

[3] Find out how you stack up to new industry benchmarks for mobile page speed). https://www.thinkwithgoogle.com / marketing-strategies / app-and-mobile / mobile-page-speed-new-industry-benchmarks/, 2018.

[4] 10 Most Popular Phones in India in 2020 – Xiaomi and Samsung Rules. https://candytech.in/most-popular-phones-in-india/, 2020.

[5] Browser Market Share Worldwide. https://gs.statcounter.com/browser-market-share, 2020.

[6] Parallel.js. https://github.com/parallel-js/parallel.js, 2020.

[7] Same-origin Policy. https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy, 2020.

[8] The 10 Best Chromium Browser Alternatives Better Than Chrome. https://www.makeuseof.com/tag/alternative-chromium-browsers/, 2020.

[9] Threads.js. https://github.com/andywer/threads.js, 2020.

[10] 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. NSDI '15. USENIX, 2015.

[11] 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. In *Proceedings of the 2016 Internet Measurement Conference*, IMC '16, page 319–325. Association for Computing Machinery, 2016.

[12] Alexa. Top Sites in the United States. http://www.alexa.com/topsites/countries/US, 2018.

[13] Amazon. Silk Web Browser. https://docs.aws.amazon.com/silk/latest/developerguide/introduction.html, 2018.

[14] D. An. Find out how you stack up to new industry benchmarks for mobile page speed. https://www.thinkwithgoogle.com / marketing-resources / data-measurement / mobile-page-speed-new-industry-benchmarks/, 2018.

[15] S. Apostolakis, Z. Xu, G. Chan, S. Campanoni, and D. I. August. Perspective: A Sensible Approach to Speculative Automatic Parallelization. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 351–367. Association for Computing Machinery, 2020.

[16] W. Aqeel, B. Chandrasekaran, A. Feldmann, and B. Maggs. On Landing and Internal Web Pages. In *Proceedings of the 2020 ACM SIGCOMM Conference on Internet Measurement Conference*, IMC. ACM, 2020.

[17] N. Bhatti, A. Bouch, and A. Kuchinsky. Integrating User-perceived Quality into Web Server Design. World Wide Web Conference on Computer Networks : The International Journal of Computer and Telecommunications Networking, 2000.

[18] A. Bouch, A. Kuchinsky, and N. Bhatti. Quality is in the Eye of the Beholder: Meeting Users' Requirements for Internet Quality of Service. CHI, The Hague, The Netherlands, 2000. ACM.

[19] M. Butkiewicz, D. Wang, Z. Wu, H. Madhyastha, and V. Sekar. Klotski: Reprioritizing Web Content to Improve User Experience on Mobile Devices. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation*, NSDI. USENIX Association, 2015.

[20] C. Cadar and K. Sen. Symbolic Execution for Software Testing: Three Decades Later. *Commun. ACM*, 56(2):82–90, Feb. 2013.

[21] C. Cascaval, S. Fowler, P. Montesinos-Ortego, W. Piekarski, M. Reshadi, B. Robatmili, M. Weber, and V. Bhavsar. ZOOMM: A parallel web browser engine for multicore mobile devices. In *PPoPP*, 2013.

[22] M. Chaqfeh, Y. Zaki, J. Hu, and L. Subramanian. JSCleaner: De-Cluttering Mobile Webpages Through JavaScript Cleanup. In *Proceedings of The Web Conference 2020*, WWW '20. ACM, 2020.

[23] Cookiebot. Google ending third-party cookies in Chrome. https://www.cookiebot.com/en/google-third-party-cookies/, 2020.

[24] 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*, IMC '18, New York, NY, USA, 2018. ACM.

[25] M. Dasari, S. Vargas, A. Bhattacharya, A. Balasubramanian, S. R. Das, and M. Ferdman. Impact of device performance on mobile internet QoE. In *IMC*, 2018.

[26] L. De Moura and N. Bjørner. Z3: An Efficient SMT Solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, page 337–340. Springer-Verlag, 2008.

[27] Drupal. Drupal - Open Source CMS. https://www.drupal.org/, 2019.

[28] E. Enge. MOBILE VS. DESKTOP USAGE IN 2019. https://www.perficientdigital.com/insights/our-research/mobile-vs-desktop-usage-study, 2019.

[29] J. Erman, V. Gopalakrishnan, R. Jana, and K. K. Ramakrishnan. Towards a SPDY'Ier Mobile Web? *IEEE/ACM Trans. Netw.*, 23(6):2010–2023, Dec. 2015.

[30] D. Etherington. Mobile internet use passes desktop for the first time, study

finds. https://techcrunch.com/2016/11/01/mobile-internet-use-passes-desktop-for-the-first-time-study-finds/, 2016.

[31] T. Everts and T. Kadlec. WPO stats. https://wpostats.com/, 2019.

[32] Facebook. Prepack. https://github.com/facebook/prepack, 2019.

[33] M. J. Freedman, E. Freudenthal, and D. Mazières. Democratizing content publication with coral. NSDI. USENIX, 2004.

[34] D. F. Galletta, R. Henry, S. McCoy, and P. Polak. Web Site Delays: How Tolerant are Users? Journal of the Association for Information Systems, 2004.

[35] Y. Geng, Y. Yang, and G. Cao. Energy-Efficient Computation Offloading for Multicore-Based Mobile Devices. In *IEEE Conference on Computer Communications*, INFOCOM, pages 46–54, 2018.

[36] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed Automated Random Testing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, page 213–223. ACM, 2005.

[37] Google. Accelerated Mobile Pages Project – AMP. https://www.ampproject.org/.

[38] Google. Chromium. https://www.chromium.org/Home.

[39] Google. Why performance matters? https://developers.google.com/web/fundamentals/performance/why-performance-matters.

[40] Google. Speed Index - WebPagetest Documentation. https://sites.google.com/a/webpagetest.org/docs/using-webpagetest/metrics/speed-index, 2012.

[41] Google. Brotli compression format. https://github.com/google/brotli, 2019.

[42] Google. Lighthouse. https://developers.google.com/web/tools/lighthouse/, 2019.

[43] A. Hidayat. Esprima. http://esprima.org.

[44] HTTP Archive. State of Javascript. https://httparchive.org/reports/state-of-javascript, 2020.

[45] J. Huang, P. Prabhu, T. B. Jablin, S. Ghosh, S. Apostolakis, J. W. Lee, and D. I. August. Speculatively Exploiting Cross-Invocation Parallelism. In *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation*, PACT '16, page 207–221, New York, NY, USA, 2016. Association for Computing Machinery.

[46] 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 *Proceedings of the 25th Annual International Conference on Mobile Computing and Networking*, MobiCom. ACM, 2019.

[47] N. Kansal, M. Ramanujam, and R. Netravali. Alohamora: Reviving HTTP/2 Push and Preload by Adapting Policies On the Fly. In *Proceedings of the 18th USENIX Conference on Networked Systems Design and Implementation*, NSDI, Berkeley, CA, USA, 2021. USENIX Association.

[48] R. Ko, J. Mickens, B. Loring, and R. Netravali. Oblique: Accelerating Page Loads Using Symbolic Execution. In *Proceedings of the 18th USENIX Conference on Networked Systems Design and Implementation*, NSDI, Berkeley, CA, USA, 2021. USENIX Association.

[49] V. Le Pochat, T. V. Goethem, S. Tajalizadehkhoob, M. Korczynski, and t. . T. s. . N. y. . . Joosen, Wouter.

[50] B. Loring, D. Mitchell, and J. Kinder. ExpoSE: Practical Symbolic Execution of Standalone JavaScript. In *Proceedings of the 24th ACM SIGSOFT International SPIN Symposium on Model Checking of Software*, SPIN 2017. ACM, 2017.

[51] D. Lymberopoulos, O. Riva, K. Strauss, A. Mittal, and A. Ntoulas. PocketWeb: Instant Web Browsing for Mobile Devices. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII. ACM, 2012.

[52] M. Madsen, B. Livshits, and M. Fanning. Practical Static Analysis of JavaScript Applications in the Presence of Frameworks and Libraries. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, page 499–509. Association for Computing Machinery, 2013.

[53] H. Mai, S. Tang, S. T. King, C. Cascaval, and P. Montesinos. A Case for Parallelizing Web Pages. In *Proceedings of the 4th USENIX Conference on Hot Topics in Parallelism*, HotPar. USENIX Association, 2012.

[54] S. Mardani, M. Singh, and R. Netravali. Fawkes: Faster Mobile Page Loads via App-Inspired Static Templating. In *Proceedings of the 17th USENIX Conference on Networked Systems Design and Implementation*, NSDI, Berkeley, CA, USA, 2020. USENIX Association.

[55] MDN. Web Workers API. https://developer.mozilla.org/en-US/docs/Web/API/Worker, 2020.

[56] M. Mehrara, P.-C. Hsu, M. Samadi, and S. Mahlke. Dynamic Parallelization of JavaScript Applications Using an Ultra-lightweight Speculation Mechanism. In *Proceedings of the 2011 IEEE 17th International Symposium on High Performance Computer Architecture*, HPCA, 2011.

[57] J. Mickens. Rivet: Browser-Agnostic Remote Debugging for Web Applications. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, USENIX ATC'12, USA, 2012. USENIX Association.

[58] J. Mickens. Pivot: Fast, Synchronous Mashup Isolation Using Generator Chains. SP '14, page 261–275. IEEE Computer Society, 2014.

[59] J. Mickens, J. Elson, and J. Howell. Mugshot: De-

terministic Capture and Replay for Javascript Applications. In *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation*, NSDI'10. USENIX Association, 2010.

[60] Monsoon Solutions Inc. Power monitor software. http://msoon.github.io/powermonitor/, 2018.

[61] Y. Na, S. W. Kim, and Y. Han. JavaScript Parallelizing Compiler for Exploiting Parallelism from Data-Parallel HTML5 Applications. *ACM Trans. Archit. Code Optim.*, 12(4):64:1–64:25, Jan. 2016.

[62] J. Nejati and A. Balasubramanian. An In-depth Study of Mobile Browser Performance. In *Proceedings of the 25th International Conference on World Wide Web*, WWW '16, pages 1305–1315. International World Wide Web Conferences Steering Committee, 2016.

[63] R. Netravali, A. Goyal, J. Mickens, and H. Balakrishnan. Polaris: Faster Page Loads Using Fine-grained Dependency Tracking. In *Proceedings of the 13th USENIX Conference on Networked Systems Design and Implementation*, NSDI, Berkeley, CA, USA, 2016. USENIX Association.

[64] R. Netravali and J. Mickens. Prophecy: Accelerating Mobile Page Loads Using Final-state Write Logs. In *Proceedings of the 15th USENIX Conference on Networked Systems Design and Implementation*, NSDI, Berkeley, CA, USA, 2018. USENIX Association.

[65] R. Netravali and J. Mickens. Reverb: Speculative debugging for web applications. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '19. ACM, 2019.

[66] R. Netravali, V. Nathan, J. Mickens, and H. Balakrishnan. Vesper: Measuring Time-to-Interactivity for Web Pages. In *Proceedings of the 15th USENIX Conference on Networked Systems Design and Implementation*, NSDI, Renton, WA, USA, 2018. USENIX Association.

[67] R. Netravali, A. Sivaraman, J. Mickens, and H. Balakrishnan. WatchTower: Fast, Secure Mobile Page Loads Using Remote Dependency Resolution. In *Proceedings of the 17th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '19, pages 430–443. ACM, 2019.

[68] R. Netravali, A. Sivaraman, K. Winstein, S. Das, A. Goyal, J. Mickens, and H. Balakrishnan. Mahimahi: Accurate Record-and-Replay for HTTP. Proceedings of ATC '15. USENIX, 2015.

[69] E. Nygren, R. K. Sitaraman, and J. Sun. The akamai network: A platform for high-performance internet applications. *SIGOPS Oper. Syst. Rev.*, 44(3):2–19, Aug. 2010.

[70] OpenSignal. Pakistan: Mobile Network Experience Report, February 2020. https://www.opensignal.com/reports/2020/02/pakistan/mobile-network-experience, 2020.

[71] Opera. Opera Mini. http://www.opera.com/mobile/mini, 2018.

[72] Opera. Opera Turbo. http://www.opera.com/turbo, 2018.

[73] A. Osmani. The Cost of JavaScript. https://medium.com/@addyosmani/the-cost-of-javascript-in-2018-7d8950fbb5d4, 2018.

[74] V. N. Padmanabhan and J. C. Mogul. Using Predictive Prefetching to Improve World Wide Web Latency. *SIGCOMM Comput. Commun. Rev.*, 26(3):22–36, July 1996.

[75] J. Park, I. Lim, and S. Ryu. Battles with False Positives in Static Analysis of JavaScript Web Applications in the Wild. In *Proceedings of the 38th International Conference on Software Engineering Companion*, ICSE '16, page 61–70. ACM, 2016.

[76] C. Petrov. 52 Mobile vs. Desktop Usage Statistics For 2019 [Mobile's Overtaking!]. https://techjury.net/stats-about/mobile-vs-desktop-usage/, 2019.

[77] G. Phillips. Smartphones vs. desktops: Why is my phone slower than my pc? https://www.makeuseof.com/tag/smartphone-desktop-processor-differences/.

[78] L. Richardson. Beautiful Soup Documentation. https://www.crummy.com/software/BeautifulSoup/bs4/doc/, 2019.

[79] V. Ruamviboonsuk, R. Netravali, M. Uluyol, and H. V. Madhyastha. Vroom: Accelerating the Mobile Web with Server-Aided Dependency Resolution. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM. ACM, 2017.

[80] K. Sen, D. Marinov, and G. Agha. CUTE: A Concolic Unit Testing Engine for C. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-13, page 263–272. ACM, 2005.

[81] S. Singh, H. V. Madhyastha, S. V. Krishnamurthy, and R. Govindan. FlexiWeb: Network-Aware Compaction for Accelerating Mobile Web Transfers. In *Proceedings of the 21st Annual International Conference on Mobile Computing and Networking*, MobiCom. ACM, 2015.

[82] A. Sivakumar, C. Jiang, S. Nam, P. Shankaranarayanan, V. Gopalakrishnan, S. Rao, S. Sen, M. Thottethodi, and T. Vijaykumar. Scalable Whittled Proxy Execution for Low-Latency Web over Cellular Networks. In *Proceedings of the 23rd Annual International Conference on Mobile Computing and Networking*, Mobicom. ACM, 2017.

[83] A. Sivakumar, S. Puzhavakath Narayanan, V. Gopalakrishnan, S. Lee, S. Rao, and S. Sen. Parcel: Proxy assisted browsing in cellular networks for energy and latency reduction. In *Proceedings of the 10th ACM Inter-*

national on Conference on Emerging Networking Experiments and Technologies, CoNEXT '14, pages 325–336, New York, NY, USA, 2014. ACM.

[84] Y. Suzuki. Estraverse. https://github.com/estools/estraverse.

[85] X. S. Wang, A. Balasubramanian, A. Krishnamurthy, and D. Wetherall. Demystifying Page Load Performance with WProf. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, NSDI. USENIX Association, 2013.

[86] X. S. Wang, A. Balasubramanian, A. Krishnamurthy, and D. Wetherall. How Speedy is SPDY? In *Proceedings of NSDI*, NSDI'14, pages 387–399, Berkeley, CA, USA, 2014. USENIX Association.

[87] X. S. Wang, A. Krishnamurthy, and D. Wetherall. Speeding Up Web Page Loads with Shandian. In *Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation*, NSDI. USENIX Association, 2016.

[88] Z. Wang, F. X. Lin, L. Zhong, and M. Chishtie. How Far Can Client-only Solutions Go for Mobile Browser Speed? In *Proceedings of the 21st International Conference on World Wide Web*, WWW '12. ACM, 2012.

[89] WordPress. Blog Tool, Publishing Platform, and CMS – WordPress. https://wordpress.org/, 2019.

[90] Y. Zaki, J. Chen, T. Pötsch, T. Ahmad, and L. Subramanian. Dissecting Web Latency in Ghana. In *Proceedings of the 2014 Conference on Internet Measurement Conference*, IMC '14, page 241–248. Association for Computing Machinery, 2014.