# WatchTower: Fast, Secure Mobile Page Loads Using Remote Dependency Resolution

Ravi Netravali
UCLA

Anirudh Sivaraman
NYU

James Mickens
Harvard University

Hari Balakrishnan
MIT CSAIL

## ABSTRACT

Remote dependency resolution (RDR) is a proxy-driven scheme for reducing mobile page load times; a proxy loads a requested page using a local browser, fetching the page's resources over fast proxy-origin links instead of a client's slow last-mile links. In this paper, we describe two fundamental challenges to efficient RDR proxying: the increasing popularity of encrypted HTTPS content, and the fact that, due to time-dependent network conditions and page properties, RDR proxying can actually *increase* load times. We solve these problems by introducing a new, *secure* proxying scheme for HTTPS traffic, and by implementing WatchTower, a *selective* proxying system that uses dynamic models of network conditions and page structures to only enable RDR when it is predicted to help. WatchTower loads pages 21.2%–41.3% faster than state-of-the-art proxies and server push systems, while preserving end-to-end HTTPS security.

**ACM Reference Format:**
Ravi Netravali, Anirudh Sivaraman, James Mickens, and Hari Balakrishnan. 2019. WatchTower: Fast, Secure Mobile Page Loads Using Remote Dependency Resolution. In *The 17th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys '19), June 17–21, 2019, Seoul, Republic of Korea.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3307334.3326104

## 1 INTRODUCTION

Loading web pages has become increasingly complex. In a modern page, the top-level HTML embeds references to dozens of objects [15], each of which must be fetched and evaluated. The evaluation of an object may cause additional objects to be fetched and evaluated, e.g., when a CSS file refers to an image, or a JavaScript file issues an AJAX request. Thus, to completely load a page, a browser must recursively resolve a complex *dependency graph* that specifies the objects to fetch, and the order in which they can be evaluated [44]. Figure 1 presents an example page and its dependency graph.

To provide the fast page loads that users demand [11, 12, 24], a browser must quickly resolve dependency graphs. However, dependencies between different objects serialize the resolution process and increase load times [44, 58], especially on mobile networks where round trip times tend to be high. Given the rapid increase in mobile web traffic [20, 63], considerable effort has been expended to accelerate the dependency resolution process and reduce mobile page load times.

One line of solutions enables servers to proactively push objects to clients in anticipation of future requests, e.g., using the HTTP/2 protocol [9]. With server push, clients can avoid incurring network round trips in the critical path of their page loads. Though promising, developing push policies is challenging, leading to mixed performance improvements and low adoption rates [21, 57, 70, 74]. The reason is that identifying the precise resources that comprise a page often requires parsing and executing the page's objects—something that existing simple push policies do not do. Furthermore, the resources on a page can change even across back-to-back loads [58], requiring additional parsing and execution.

A more promising approach to generate server push policies, embodied in both academic prototypes [10, 48, 62, 71] and commercial systems [5, 36, 52], is to use *remote dependency resolution (RDR) proxies.* An RDR proxy resides on a cloud server with low-latency network paths to origin servers. To load a page, a client browser does not send individual HTTP requests to the RDR proxy; instead, the client browser only sends the request for a page's top-level HTML. The proxy loads the requested page locally with a headless browser (i.e., one without a GUI [32]), streaming the discovered objects to the client immediately. In this way, RDR proxies resolve an entire dependency graph using low-latency proxy links. RDR proxies can perform additional proxy-based optimizations like caching [73] and compression [2, 53, 60], which provide orthogonal benefits (e.g., compression is useful for mobile clients with limited data plans), but have limited impact on load times.[1] In addition, RDR proxies overcome the limitations that server push policies have with respect to identifying the resources that clients need to load a page [61].

Unfortunately, two challenges limit the extent to which RDR proxies can reduce load times. The first challenge is that **RDR proxies must either ignore HTTPS traffic, or compromise the end-to-end security of HTTPS.** An increasing fraction of web traffic uses HTTPS instead of HTTP [29, 41]. A core goal of HTTPS is end-to-end security: only web servers and browsers

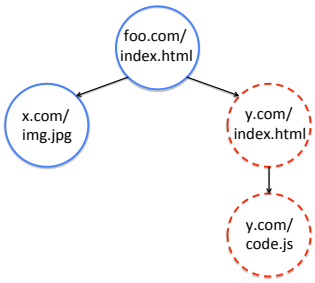

*MobiSys '19, June 17–21, 2019, Seoul, Republic of Korea*

ACM ISBN 978-1-4503-6661-8/19/06…$15.00
https://doi.org/10.1145/3307334.3326104

[1] Caching and compression proxies (like content delivery networks [4, 23, 50, 68]) operate on individual objects, so clients still must traverse the high-latency last-mile links to proxies for each resource in a page, limiting load time benefits [2].

```
<html>
  <img src=``http://x.com/img.jpg''/>
  <iframe src=``https://y.com/index.html''>
    <!--Inside the frame . . .-->
    <script src=``https://y.com/code.js''/>
  </iframe>
</html>
```

(a) The page contains two HTTP objects and two HTTPS objects.




(b) The dependency graph. Blue circles indicate HTTP content, and dashed red circles indicate HTTPS content.

**Figure 1: The web page `http://foo.com/index.html`, and its dependency graph.**

should be able to decrypt network data, with intermediate hops learning nothing about the forwarded ciphertext [56]. These semantics present a dilemma for third-party proxies (e.g., belonging to a mobile carrier). A proxy can ignore HTTPS traffic and force browsers to directly resolve HTTPS dependency graphs, but this denies load acceleration to a growing fraction of pages. Alternatively, a proxy can act as a man-in-the-middle for HTTPS traffic [17, 34, 55] to make the associated pages load faster. However, such full dependency resolution violates HTTPS' end-to-end security guarantees.

The second challenge is that **remote dependency resolution can actually increase page load times.** Traditionally, web proxies are "always-on," i.e., a browser always uses the proxy for each page load. However, the ability of a proxy to improve load times depends on subtle interactions between a page's dependency graph and the latencies between the browser, the proxy, and a page's web servers. Network conditions change over time; clients, proxies, and web servers may change locations; and no two pages have the same dependency graph. Thus, at any given moment, for any given page, a browser may increase load times by using remote dependency resolution.

This paper investigates two important questions raised by these observations. First, is the tension between security and performance resolvable in practice? Second, can a client automatically determine when to invoke remote dependency resolution? Through experiments with many real pages (Alexa top 500 landing pages [3], and smaller sets of interior and personalized pages), live mobile networks (cellular, residential WiFi, and in-flight WiFi), and multiple client devices (phones, laptops, desktops), we answer these questions affirmatively, and make three primary contributions.

**Balancing security and performance with HTTPS sharding:** First, we quantify the performance penalties that RDR proxies incur by ignoring HTTPS traffic (§4). We find that HTTP-only proxying achieves less than 60% of the benefits enjoyed by a "full" proxy that resolves both HTTP objects and HTTPS objects. To overcome this, we introduce a new proxy technique called *HTTPS sharding*, which allows the HTTPS subgraphs in a page to be resolved remotely, in a way that protects end-to-end security guarantees (§4). HTTPS sharding gives each HTTPS origin the ability to proxy its own parts of a dependency graph. Our key observation is that dependency graphs often contain connected subgraphs, where each subgraph's HTTPS vertices all belong to the same origin. This, in turn, limits the number of round trips that clients must incur to each origin's proxy. As a result, HTTPS sharding provides over 90% of the benefits of full dependency resolution, without exposing sensitive HTTPS objects from origin $X$ to proxies belonging to a different origin $Y$.

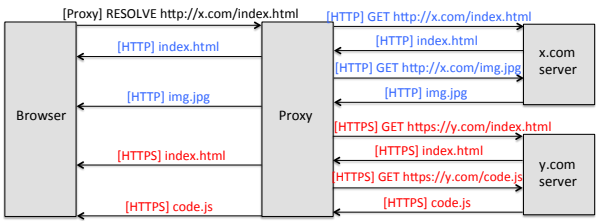
**Model-based selective proxying:** Second, we quantify the performance penalties incurred by RDR proxies that use an always-on model (§5). For example, we find that, on a residential WiFi network, HTTP-only RDR hurts 37.8% of page loads. To decrease the likelihood that RDR proxies do harm, we introduce a *model-based selective proxying* algorithm (§6.1). Clients can use this algorithm to predict load times for a page when using a proxy, or when loading the page directly. Our predictive models take as inputs a page's dependency graph, the dependency resolution scheme, and the estimated network latencies that would be incurred by traversing the graph with or without a proxy. Surprisingly, our models make highly accurate predictions of whether or not to use a proxy (error rates of 0.61%–1.23%), without considering several details of the page load process that are hard to model, such as browser computation delays and TCP congestion control.

**WatchTower: a new RDR proxy system that is fast and secure:** Finally, we implement and evaluate WatchTower, a new proxy system that uses both HTTPS sharding and selective proxying to outperform a traditional always-on proxy, various server push policies, and Vroom [58], a state-of-the-art mobile web accelerator (§6). For example, on a cellular network, WatchTower outperforms an always-on proxy and Vroom by over 29% and 34%, respectively, resulting in page load time reductions of 1.22–1.85 *seconds*, and median energy savings of 22%. WatchTower's benefits extend to other performance metrics like Speed Index, with speedups of 26% on a cellular network.

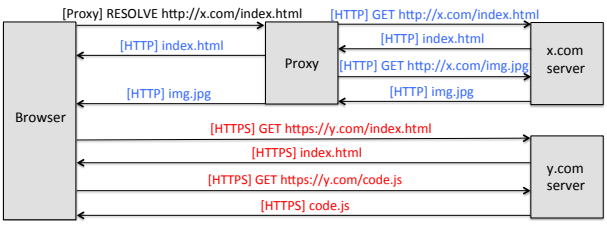
We evaluate WatchTower in various additional scenarios, including with partial adoption, warm client-side caches, and stale model inputs (§6.3). We also discuss operational overheads (§6.3) and potential deployment avenues (§4) for WatchTower and HTTPS sharding.

## 2 BACKGROUND AND RELATED WORK

**Remote dependency resolution:** RDR is used by a variety of web accelerators. For example, Amazon Silk [5], Opera Mini [52], and CGN [10] perform **full dependency resolution**, using proxy servers to resolve both HTTPS and HTTP edges in a dependency graph. Figure 2a provides a high-level overview of full RDR. As mentioned in §1, full RDR breaks the end-to-end security guarantees of HTTPS. PARCEL [62] avoids this issue by using a proxy to only resolve the dependency graph's HTTP edges. However, such




**(a) Full dependency resolution. The user sends a page's top-level URL to the proxy. The proxy uses a headless browser to resolve the dependency graph (which contains both HTTP and HTTPS objects), and sends those objects back to the user's browser.**




**(b) HTTP-only dependency resolution. As with full RDR, the user sends a page's top-level URL to the proxy. However, the proxy can only evaluate the HTTP edges in the dependency graph. The proxy returns the fetched HTTP objects to the user. The user's browser must evaluate the remaining, HTTPS part of the dependency graph.**

**Figure 2: Comparing full and HTTP-only RDR for the page in Figure 1. "[XXX]" prefaces network traffic sent using the XXX protocol. Blue and red text indicate HTTP and HTTPS traffic, respectively. Note that, for both resolution techniques, the proxy pipelines results back to the client, returning an object as soon as the proxy has fetched it.**

**HTTP-only dependency resolution** forces browsers to fetch HTTPS content directly from origin servers, over slow last-mile links (Figure 2b). HTTPS traffic is rapidly overtaking HTTP traffic [29, 41], resulting in declining benefits for HTTP-only RDR (§4). Even today, Google Chrome reports that over 70% of page loads use HTTPS, with 96 out of the top 100 pages in the world supporting HTTPS [29].

Certain RDR systems like Shandian [71], Prophecy [45], and Opera Mini [52], return post-processed versions of objects to clients. Such content alteration can reduce client-side computation and bandwidth costs, but is fragile and may break page functionality [2]. WatchTower is orthogonal to such approaches; to the extent that computationally-optimized pages still require multiple round trips to load, WatchTower can hide the associated network latencies. Moreover, our experimental results show that, even though computational overheads can be significant on mobile devices [43, 58, 70, 72], WatchTower still significantly reduces mobile page load times. WatchTower is also compatible with RDR optimizations that employ whittling techniques to reduce the proxy-side overheads of discovering resources to push [61].

**Compression proxies:** Both full and HTTP-only RDR require proxies to run a headless browser—only by running this browser and simulating a client-side page load can a proxy generate the dependency graph to resolve. Compression proxies are much simpler. A user's browser generates the dependency graph locally, but forwards individual fetch requests to the compression proxy. The proxy downloads the associated objects and returns compressed versions to the browser. Large technology companies often run compression proxies to reduce the bandwidth consumption of their customers. For example, Google Flywheel [2], Opera Turbo [53], and Nokia Xpress [66] use proxies to transcode images into the WebP format [27], gzip text-based content like HTML, and minify JavaScript and CSS. Flywheel and Opera Turbo only proxy HTTP requests, while Xpress proxies HTTPS traffic too, decrypting, compressing, and then re-encrypting that traffic [39]. More recently, Flexiweb [60] lets clients make network-aware decisions about whether sending individual requests to a compression proxy will result in load time benefits.

## 3 EXPERIMENTAL METHODOLOGY

Here, we describe the setup we used to evaluate traditional proxy schemes (§4–§5) and our new system that leverages selective proxying and HTTPS sharding (§6).

**Performance baseline:** This paper focuses on reducing mobile page load times. Thus, we evaluated proxies that used remote dependency resolution (not compression proxies which have limited impact on load times [2]). The performance baselines were the load times that resulted from using a compression-only proxy, or no proxy at all. The baseline compression proxy did not perform caching, since caching proxies are most useful for static objects that are already well-served by CDNs [4, 23, 50, 68] located close to proxies.

We extended Cumulus, a state-of-the-art proxy [48], to implement our baseline proxy and the various RDR proxies that we evaluate in this paper. Cumulus has two components: a caching daemon which runs on a user's local machine, and a remote proxy which runs on a cloud server. The client browser is unmodified—Cumulus uses destination NAT filters [14] to redirect the browser's network traffic to the local caching daemon. The caching daemon maintains a database that maps HTTP(S) requests to the corresponding responses. When the browser issues a request that misses in the cache, the caching daemon forwards the request to the remote proxy. The remote proxy uses a headless browser to fetch the associated content (and anything that can be reached by recursively evaluating the fetched objects). To mimic the client-side page load and ensure that the remote proxy downloads the same object versions from the origin servers as the client would have, the headless browser reuses the HTTP headers (e.g., User-Agent) included in the client's initial request. The remote proxy ships objects back to the caching daemon as soon as they are fetched. We note that objects remain in the caching daemon for at least the duration of the corresponding page load. Objects marked as uncacheable by HTTP caching headers [1, 22] are evicted by the caching daemon immediately after the page load, and browsers employ default caching logic for all objects received from the caching daemon. Our calibration experiments [47] show that Cumulus' performance is competitive with that of other state-of-the-art proxies.

| Client network | Link Rate (Mbits/s) | First-hop RTT (ms) |
|---|---|---|
| JetBlue In-flight WiFi [35] | 7.2 | 966 |
| Verizon 4G LTE Cellular | 9.3 | 87 |
| Residential WiFi | 22.7 | 21 |
| Wired Broadband | 772 | 0.37 |

**Table 1: The real networks that we used to evaluate RDR. The client for the cellular and residential WiFi networks was a Nexus 5 phone (Android 5.1.1); mobile versions of pages were used if they existed. In-flight WiFi and wired network tests used a 2015 Macbook Pro laptop (macOS 10.14, 4 processors, 16 GB RAM) and Lenovo M91p desktop (GNU Linux 14.04, 8 processors, 8 GB RAM), respectively.**

Web sites use cookies [7] to store small amounts of client-side data associated with individual origins. When a Cumulus-enabled browser issues an HTTP request for a resource from origin $X$, the browser sends the HTTP `Cookie` headers for $X$ to the proxy. This policy, which is common in deployed proxies like Amazon Silk [5], allows a proxy to fetch customized content from origin $X$. Due to space constraints, we defer a full discussion of the trade-offs between cookie disclosure policies and proxy performance to a technical report [47].

**Evaluation setup:** Our test corpus primarily consisted of the Alexa top 500 pages [3]; we also consider non-landing and personalized pages in §6.3. Page loads with mobile devices used mobile-optimized versions of pages when available. We used Google Chrome v63 with Cumulus as the proxy. We defined "load time" as the time between the `navigationStart` and `loadEventEnd` JavaScript events; results in §6.3 also consider the Speed Index metric [26]. Results used cold browser caches unless otherwise specified.

Table 1 describes the last-mile links and client devices used in our experiments. We primarily focus on the common case of proxy-based web acceleration for mobile browsers [2, 52, 62], but also present results with laptops and desktops to show that proxies can improve load times for a variety of clients. Client devices were always physically located in Massachusetts, except for in-flight WiFi experiments. To vary network latencies between clients, proxies, and origin servers, we ran proxies on EC2 machines in northern California, Oregon, Virginia, and Brazil, as well as on a desktop in Massachusetts.

## 4 THE COST OF IGNORING HTTPS

In this section, we empirically demonstrate why HTTP-only proxies are increasingly untenable. Since this section focuses on the cost of ignoring HTTPS, not the cost of suboptimal proxy location (§5), we loaded each page five times for each of the five proxies, and recorded the lowest median load time across all proxies.

**Full vs. HTTP-only Resolution:** Figure 3 describes load times when clients used full dependency resolution, HTTP-only dependency resolution, compression-only proxying, or no proxying at all (i.e., a default browser); note that all proxies compressed objects. As expected, both forms of RDR outperformed the compression-only proxy and the default browser. Even on the high-bandwidth, low-latency broadband network, full and HTTP-only RDR decreased load times by 22.4% and 13.7%, respectively, compared to the compression-only proxy. Our results also show that with respect to load time, a compression-only proxy is essentially equivalent to no proxy. Thus, in the rest of the paper, we do not show results for a compression-only proxy; instead, our baseline is a default browser with no proxy.

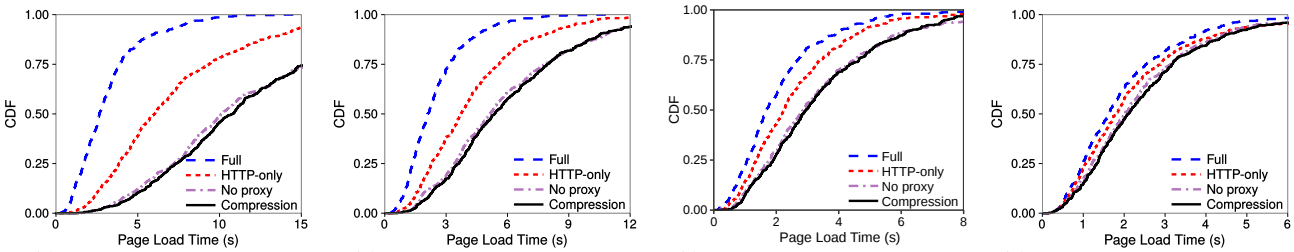
The most interesting trend is the performance difference between full and HTTP-only RDR. The Internet-wide shift from HTTP to HTTPS is ongoing, *but even today*, HTTP-only RDR is losing efficacy due to its inability to handle HTTPS traffic. As Figure 3 shows, full dependency resolution provides 1.64×–1.99× the benefits of HTTP-only resolution. On a cellular network, this performance advantage equates to an additional 1.61 seconds of load time removed at the median. The performance disparities between full and HTTP-only RDR will only increase as more sites transition to HTTPS [29, 41]. Indeed, for a site that only uses HTTPS, HTTP-only resolution can provide no benefit at all.

The advantages of full dependency resolution persist even with *warm browser caches*. For example, we loaded each page in our corpus twice, back to back, over a cellular network. For each page, we measured the time for the second load (which used a warm browser cache). Full dependency resolution decreased the median load time by 30.1%; HTTP-only resolution decreased the median load time by just 13.3%. Analyzing our test corpus revealed that only 38.7% of objects were cacheable; thus, page loads with warm caches still required most objects to be fetched. Given the superiority of full dependency resolution for both warm and cold caches, ignoring HTTPS traffic seems unjustifiable.

**HTTPS-sharding Resolution:** To allow the secure proxying of HTTPS content, we propose HTTPS-sharding resolution. In this scheme, each HTTPS origin runs its own RDR proxy, thereby preserving the end-to-end security guarantees of HTTPS. The proxy for `foo.com` only resolves HTTPS dependencies for `foo.com` objects. However, the proxy can resolve HTTP dependencies from *any* origin, since those dependencies do not involve secure data. Mobile providers can still run origin-agnostic, HTTP-only proxies. Figure 4 shows how this would work for the page in Figure 1a. Compared to full RDR (Figure 2a), the end-to-end page load incurs an extra RTT over the last mile. However, each round trip to an origin's HTTPS proxy lets the client resolve an entire HTTPS subgraph belonging to that origin.

Intuitively, HTTPS sharding leverages the fact that objects from a particular origin are often clustered into separate subgraphs in a page's dependency graph. This limits the number of times a client must contact each origin's proxy. For example, a parent iframe from `https://foo.com` may embed a child frame from `https://bar.com`. In this case, `foo.com`'s proxy would load `foo.com` objects in the parent frame, but would refuse to load the `bar.com` frame. The user's browser would then be responsible for contacting the `bar.com` proxy, which would resolve all of the edges in the `bar.com` subgraph. In this manner, `bar.com` never sees cleartext HTTPS data from `foo.com`, and vice versa.

Figure 5 compares the performance of full, HTTPS-sharding, and HTTP-only RDR. In the HTTPS-sharding case, the client assigned an HTTPS proxy to origin $O$ by selecting the proxy with the lowest latency to $O$'s origin servers. With HTTP-only resolution, proxies achieve only 50.3%–59.2% of the gains provided by full resolution.




(a) In-flight WiFi network. (b) Verizon 4G LTE network. (c) Residential WiFi network. (d) Wired broadband network.

**Figure 3: Distribution of page load times using full dependency resolution, HTTP-only resolution, compression-only proxying, and no proxying (i.e., a default browser).**

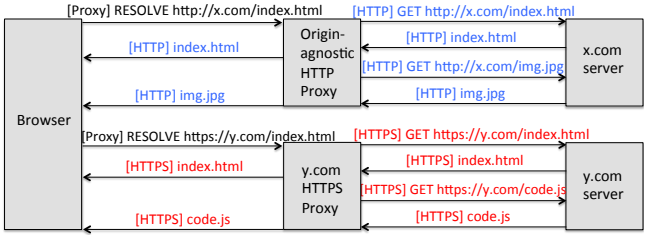



**Figure 4: With HTTPS-sharding resolution, the user's browser contacts two different proxies to load the example page. The HTTPS origin's data is never exposed to the HTTP-only proxy. Blue and red text indicate HTTP and HTTPS traffic, respectively, and proxies use pipelining to immediately return objects to the browser.**

With HTTPS-sharding, proxies achieve 91.3%–95.6% of the benefits, while preserving the end-to-end security of HTTPS.

**Deployment and Adoption:** There are several potential deployment paths for configuring and running an HTTPS-sharding proxy. For example, an origin can use Apache's reverse proxy support [6] to run an HTTPS-sharding proxy on the same machine that hosts the origin's real web server. Alternatively, a company with geographically distributed web servers may choose to strategically deploy proxies at a smaller number of vantage points. Middlebox approaches for improving performance [2, 5, 73], security [13, 18], and load balancing [31, 49] are already common, so deploying proxies is already within the capability of many companies.

Individual origins must explicitly opt into HTTPS sharding by running proxies. The performance benefits of HTTPS sharding obviously increase as more origins run proxies, but within a single page, only some HTTPS origins may run proxies. We evaluate the performance of HTTPS sharding with partial deployments in §6.3.

# 5 THE COST OF ALWAYS-ON PROXIES

Section 4 showed that remote dependency resolution can substantially decrease page load times. However, this general result hides the fact that, in some cases, RDR proxies *increase* load times. Here, we evaluate the extent to which four factors affect the benefits of RDR: the complexity of a page's dependency graph (§5.1), last-mile latency (§5.1), last-mile bandwidth (§5.3), and proxy location (§5.2). These empirical results motivate the design of our selective proxying system that avoids the pitfalls of an always-on approach (§6).

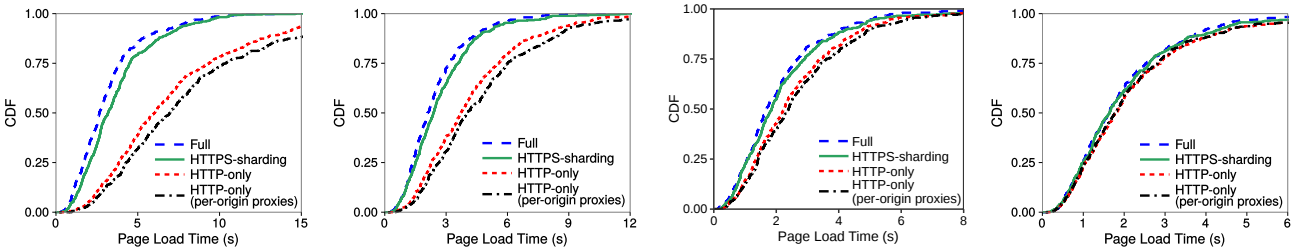
In this section, we measure the speedup that proxying confers, relative to a browser that uses no proxy. "Speedup" is defined as the ratio of page load time for a browser using no proxy, compared to the load time (under identical network conditions) for a browser using Cumulus. Speedups greater than 1 occur when RDR decreases load times. All page loads used a browser in Massachusetts with a cold cache. Due to space limitations, we sometimes only present graphs for full dependency resolution, but all reported trends are exhibited by the other resolution techniques.

## 5.1 Last-mile Latency & Page Complexity

As discussed in Section 2, remote dependency resolution provides the most benefits when the client's last-mile link has high latency. Figure 6, which plots speedups with full dependency resolution, reaffirms this point. However, for a given network, different sites experience vastly different speedups. In fact, some sites experience a slowdown by using a proxy, even with high last-mile latencies. For example, speedups were less than 1 for 3.1% and 14.3% of pages loaded over the cellular and residential WiFi networks, respectively.

To understand why different sites see different speedups for the same network conditions, consider `https://www.google.com` and `http://www.tmz.com/`. Figure 7 shows the dependency graphs for these pages. Define the *critical path* as the longest chain of non-parallelizable network fetches. The length of the critical path is a good approximation of the effort needed to load a page when latency is the limiting network factor [44, 69]. The Google page has a critical path of length four, while TMZ's has length ten. Thus, for a regular, proxy-less browser, increased last-mile latency will hurt TMZ page loads more than Google page loads.

To demonstrate this phenomenon, we used a synthetic network setup to isolate the impact of last-mile latency. Our experiment leveraged Mahimahi [48], a tool for recording and replaying HTTP traffic over emulated networks. Our simple network contained three links: one connecting the client to the proxy, another connecting the proxy to origin servers, and one connecting clients to origins. All link rates were 12 Mbits/s, but we conducted a parameter sweep of $RTT_{client-origin}$ (i.e., the RTT between the client and origins), examining values from 0 ms to 300 ms. For each tested value, we loaded each page twice: once with no proxy, and once with a proxy that did full dependency resolution, and had $RTT_{client-proxy} = RTT_{client-origin}$ and $RTT_{proxy-origin} = 0$. By setting $RTT_{proxy-origin}$ to 0, we modeled an ideal proxy that could fetch objects with no delay. Thus, the benefits of remote dependency resolution could only be influenced by last-mile latency and the structure of a page's dependency graph.




(a) In-flight WiFi network. (b) Verizon 4G LTE network. (c) Residential WiFi network. (d) Wired broadband network.

**Figure 5: Page load times using full dependency resolution, HTTPS-sharding resolution, and two variants of HTTP-only resolution (one in which a client uses a single proxy to download all HTTP content, and one in which a client downloads HTTP content from origin $O$ using the closest proxy to $O$'s servers). The per-origin HTTP-only scheme performs slightly *worse* than the standard HTTP-only scheme, incurring an extra RTT when HTTP content from origin $X$ embeds HTTP content from origin $Y$; such cross-HTTP-origin transitions are common. Thus, we do not discuss per-origin HTTP-only proxying in the rest of the paper.**

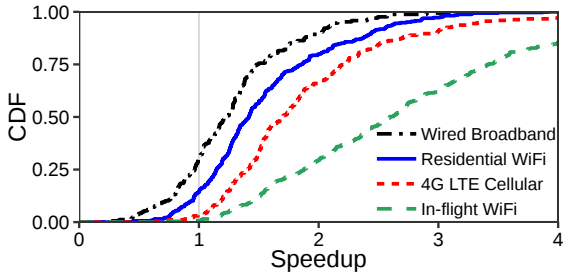
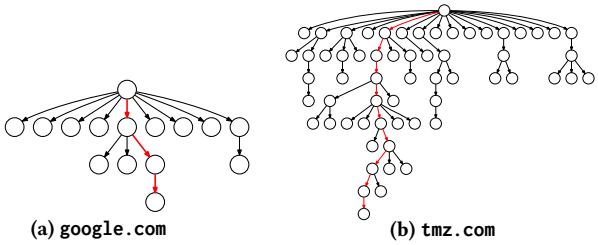



**Figure 6: Speedups for full dependency resolution. Each page load used the best possible proxy in California, Oregon, Virginia, Massachusetts, or Brazil.**



(a) `google.com` (b) `tmz.com`

**Figure 7: Dependency graphs for the Google and TMZ (cropped) homepages. Critical paths are marked in red.**

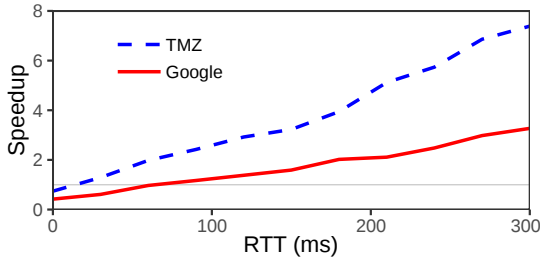
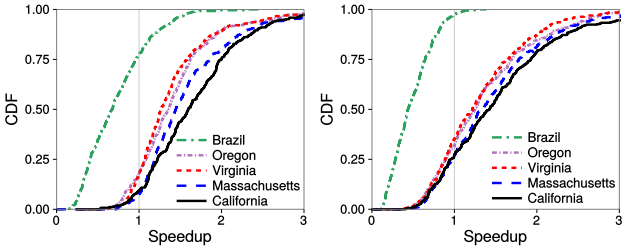



**Figure 8: Speedups increase as last-mile latency grows or critical path length increases. These results used full RDR, but the trends are similar for HTTP-only and HTTPS-sharding. The client was a Nexus 5 phone.**

As Figure 8 shows, RDR provided more benefit to the TMZ page, which had a longer critical path. However, for both pages, the proxy




(a) Verizon 4G LTE. (b) Residential WiFi network.

**Figure 9: Impact of proxy location for a client in Massachusetts. The proxy used full RDR. Results were similar for in-flight WiFi and wired broadband.**

*increased* load times when last-mile latency was low; for TMZ, the break-even point was 20 ms, but for Google, the break-even point was 65 ms. This is explained by two observations:

- As critical path length grows, proxying becomes advantageous even for small round-trip latencies, since, without proxying, chains of small latencies on the critical path sum to a large, non-parallelizable latency.
- With proxying, a page's HTML, CSS, JavaScript, and images must be parsed and evaluated twice (once on the headless browser, and once on the client browser). As the computational cost for evaluating a page increases, the double computation penalty of proxying grows. Thus, round-trip latencies must be sufficiently large to overcome the double computation penalty.

These results (from a mobile phone) show that, despite the fact that computational overheads can be significant on mobile devices [43, 58], critical path length (and the associated serial chain of last-mile latencies) is usually the primary determinant of whether a proxy will decrease mobile load times. Since the TMZ page's critical path was longer than Google's, the TMZ page started to benefit from proxying at lower last-mile latency values.

## 5.2 Proxy Location

A compression-only proxy has two goals: reduce the bandwidth needed to fetch an object, while adding minimal latency. To reduce the added latency, compression proxies use geographic affinity between clients and proxies—a client preferentially attaches to a

nearby proxy, minimizing the network detours from the natural fetch path [2, 60]. Client-driven geographic affinity is a good idea for compression-only proxies, *but a bad idea for dependency resolution proxies*. The latter type of proxy only helps if the proxy has low-latency paths to origin servers, i.e., if resolving edges in the dependency graph is faster using the proxy's links instead of the client's last-mile link. Thus, dependency resolution proxies should use *origin-driven* geographic affinity—a client which needs to load a page from origin $X$ should select a proxy which is close to $X$'s web servers.

To evaluate the importance of proxy location, we fixed the client location to Massachusetts, and loaded our 500 test pages using the four networks from Table 1. For each page and each network, we measured six median load times: a baseline load which used no proxy, and five instances of full dependency resolution using the proxies described in §3. Each median was computed over five page loads that used the same proxy. As shown in Figure 9, proxy location has a significant impact on the performance of remote dependency resolution. For example, on the cellular network, median speedups ranged from 0.68×–1.60× across the five proxy locations. In all of the networks, the biggest speedups resulted from using the proxies in California and Massachusetts. With GeoIP [37], we found that web servers in California and Massachusetts hosted 84.2% of all objects in our test pages (roughly 42% each).[2] This result shows the importance of origin-driven proxy placement, and confirms results from prior smaller-scale tests [10].

### 5.3 Last-mile Bandwidth

Prior work has shown that load times are only sensitive to network bandwidth when bandwidth is very low [2, 8, 38, 64]. We validated that RDR is also largely insensitive to link rates. This result is unsurprising, so we defer a full discussion to a technical report [47].

### 5.4 Summary

The results in this section challenge conventional wisdom, demonstrating that always-on proxies can *increase* page load times. Broadly speaking, remote dependency resolution is more likely to hurt load times when last-mile latencies are low, critical path lengths are small, or proxies are far from origin servers. However, for a particular client at a particular time, the usefulness of RDR is a function of the current network conditions, the page to load, and the available proxies.

## 6 WATCHTOWER

WatchTower is a new RDR system that performs selective proxying. When a user tries to load a page, WatchTower consults analytical models to predict whether RDR proxying would actually increase the load time; if so, WatchTower loads the page by fetching objects directly from origin servers. By making such predictions for each page load, WatchTower improves load times by 21.2%–41.3% compared to always-on proxies and state-of-the-art server push systems. Interestingly, WatchTower unlocks these benefits using simple analytical models that consider network latencies and dependency graph structures, but ignore TCP windows, computational costs, and other dynamics which might seem essential to capture.

[2] 92% of objects served in Massachusetts were from CDNs.

### 6.1 Estimating Page Load Times

A page's static critical path is the longest root-to-leaf path in the dependency graph, where "length" is defined as "the number of edges." If each edge requires the same amount of time to resolve, the static critical path provides a lower bound on the time required to load the entire page [69]. However, due to the network conditions at the time of a page load, the path with the most edges may not be the slowest one to resolve [44]. For example, a path that traverses many low-latency network links may be faster to resolve than a path that traverses a few high-latency links. Thus, our analytical models estimate the resolution time for a page's *dynamic* critical path, i.e., the path that, given current network conditions, will lower-bound a page's load time.

At a high level, we find the dynamic critical path by considering each path in the dependency graph, and estimating the resolution time for that path *as if the graph consisted of just that path*. The type of dependency resolution that a client employs (full, HTTP-only, HTTPS-sharding, or none) determines the resolution cost of each path. We then estimate the page's load time as the longest resolution time for an individual path.

Define a *context transition* as a dependency edge from HTTP content to HTTPS content, or from HTTPS to HTTP content, or from HTTPS content in origin $X$ to HTTPS content in origin $Y$. For example, Figure 1 depicts a single context transition. If proxies do not use full dependency resolution, then context transitions may inject additional network latencies into the resolution process. Our models ignore computational costs and link bandwidths since those factors are less important than link latency and page complexity (§5.1 & §5.3). Below, we describe how we estimate path resolution costs for each resolution approach.

**No remote dependency resolution:** All edges in a path are resolved using the client's network links to origin servers. As a result, the path's load time is $L * RTT_{client-origin}$, where $L$ represents the path length.

**Full dependency resolution:** We estimate load time as $RTT_{client-proxy} + (L * RTT_{proxy-origin})$. This represents the costs for the proxy to resolve the path, and the client to fetch the associated objects from the proxy.

**HTTP-only resolution:** In this scheme, an HTTP-to-HTTPS transition forces the proxy to halt resolution; the proxy returns the HTTP objects that it has fetched to the client, and the client starts to resolve HTTPS edges in the path. If the client encounters an HTTPS-to-HTTP transition, the client forwards the transition-triggering fetch to the proxy. The proxy resolves HTTP edges until hitting an HTTP-to-HTTPS transition.

Let $H$ be the number of HTTP edges in the path to resolve. The $i$th HTTP-to-HTTPS transition completes a round-trip of $RTT_{client-proxy}$, since a proxy must terminate resolution and push the fetched HTTP objects back to the client. Before the transition, the proxy also incurred fetch latencies of $H_i * RTT_{proxy-origin}$, where $H_i \leq H$ is the number of HTTP edges that the proxy resolved before the current transition (but after any previous transition). Thus, resolving the HTTP edges between two transitions incurs a latency of $RTT_{client-proxy} + (H_i * RTT_{proxy-origin})$.

Let $T$ be the number of HTTP-to-HTTPS transitions, and let $S$ be the number of HTTPS edges in the path. Resolving all of the

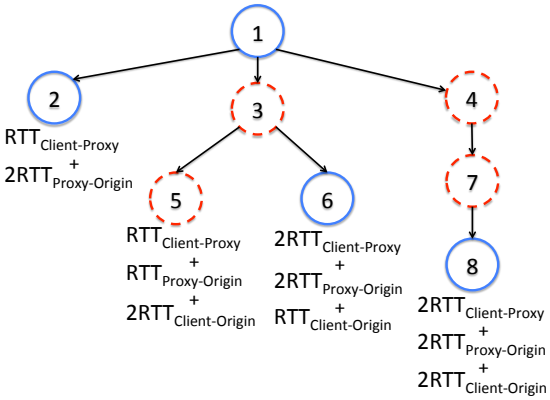



**Figure 10: Estimating the load time for a simple page. Blue circles represent HTTP objects, and dashed red circles represent HTTPS objects. Beneath each path, we list the estimated load time with HTTP-only resolution. The page's load time is estimated as the slowest resolution time amongst all paths.**

HTTPS edges incurs a total latency of $S * RTT_{client-origin}$. So, the total latency for HTTP-only resolution is the cost to resolve all of the HTTP and HTTPS edges. This cost is $S * RTT_{client-origin} + \sum_i (RTT_{client-proxy} + (H_i * RTT_{proxy-origin}))$, which equals $S * RTT_{client-origin} + T * RTT_{client-proxy} + H * RTT_{proxy-origin}$.

**HTTPS-sharding resolution:** With HTTPS-sharding resolution, all objects are fetched by proxies. The proxy for origin $X$ can fetch HTTPS objects from $X$, as well as HTTP objects from any origin. Let $\text{HTTP}_X$ or $\text{HTTPS}_X$ represent an HTTP or HTTPS object from origin $X$, and let $*$ represent any origin. There are four possible context transitions:

- In an $\text{HTTP}_X$-to-$\text{HTTPS}_X$ transition, the proxy for $X$ can fetch the needed HTTPS object.
- In an $\text{HTTP}_X$-to-$\text{HTTPS}_Y$ transition, the proxy for $X$ cannot fetch the HTTPS object from $Y$. So, the proxy must terminate dependency resolution and return any fetched objects to the client, who is then responsible for contacting the proxy for $Y$. Let $T_{HTTP_X-to-HTTPS_Y}$ represent the number of such context transitions.
- An $\text{HTTPS}_X$-to-$\text{HTTPS}_Y$ transition similarly requires the proxy for $X$ to terminate resolution. Define $T_{HTTPS_X-to-HTTPS_Y}$ as the number of these transitions.
- An $\text{HTTPS}_X$-to-$\text{HTTP}_*$ transition does *not* force $X$'s proxy to terminate resolution, since $X$'s proxy can fetch HTTP objects from any origin.

Since all objects are fetched by proxies, the minimum latency for resolving the path is $RTT_{client-proxy} + (L * RTT_{proxy-origin})$; this represents the case in which there are no $\text{HTTP}_X$-to-$\text{HTTPS}_Y$ or $\text{HTTPS}_X$-to-$\text{HTTPS}_Y$ transitions. The general formula which accounts for those transitions is $(1 + T_{HTTP_X-to-HTTPS_Y} + T_{HTTPS_X-to-HTTPS_Y}) * RTT_{client-proxy} + (L * RTT_{proxy-origin})$. In other words, each transition incurs a cost of $RTT_{client-proxy}$ when a proxy must terminate resolution.

**Selective proxying:** A client should only use a proxy if the proxy-mediated page load would be faster than an unassisted one. Using the formulas above, we can predict the two load times (Figure 10), and enable proxying only if it would provide benefits.

| Client network | HTTP-only | HTTPS-sharding |
|---|---|---|
| 4G LTE Cellular | 1.22 (5.54) | 1.85 (7.45) |
| Residential WiFi | 0.57 (1.75) | 0.81 (3.46) |
| Wired Broadband | 0.20 (0.93) | 0.39 (1.42) |

**Table 2: Median (95th percentile) seconds of load time shed by WatchTower compared to always-on proxying.**

A client may be able to choose from multiple proxies, and a page may contain objects from many origins. In these scenarios, WatchTower's formulas use the appropriate $RTT_{client-proxy}$ and $RTT_{proxy-origin}$ values. Also note that, if a client or proxy already has a cached version of an object, we set the resolution time for the object's dependency edge to zero.

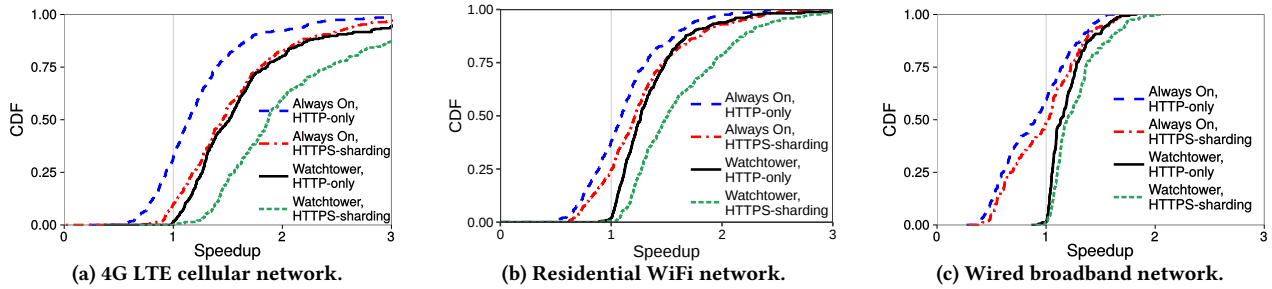
## 6.2 Design of WatchTower

WatchTower consists of two parts: a browser plugin which allows users to select a dependency resolution mode,[3] and a set of Cumulus proxies that are scattered throughout the Internet. A proxy may be origin-agnostic (as most proxies are today) and deployed by a mobile carrier or an ISP to improve load times for all customers and all pages. Alternatively, a proxy may be run by a specific origin for the purpose of supporting HTTPS-sharding resolution; such a proxy lets the origin protect the end-to-end security of its HTTPS traffic.

In WatchTower, the Cumulus proxies periodically measure their latencies to the 500 most popular origin servers from the Alexa list [3]. The proxies also capture dependency graphs for the 500 most popular sites. The WatchTower browser plugin periodically contacts each proxy, downloading the latencies and dependency graphs collected by the proxy. The plugin also measures client latencies to the Cumulus proxies, and to the 500 most popular origins. With estimates for $RTT_{client-proxy}$, $RTT_{client-origin}$, and $RTT_{proxy-origin}$, the plugin uses the formulas from §6.1 to determine, for each page load, whether the browser should fetch all objects directly, or employ the user-selected proxying technique. In the latter case, WatchTower uses its models to select the *best* proxy to use.

Of course, a user may visit sites that are not among the 500 most popular pages. When a user's WatchTower plugin sees the first request for such a page, the plugin defaults to using the closest proxy to load the page. Although client-driven geographic affinity limits the *gains* of RDR proxies (§5.2), selecting a nearby proxy for the first load of a page minimizes network indirection, and thus bounds the potential *harm* imposed by the proxy. Upon receiving the request, the proxy loads the page, streaming objects to the client and discovering the origins referenced by the page's content. The proxy adds the discovered origins to the set of origins that the proxy probes for latency. Meanwhile, as the client loads the page, the WatchTower plugin extracts the dependency graph for the page by observing initiation contexts for object fetches [44]. Later, when the client periodically polls the other proxies, the client informs the proxies about the long-tail origins belonging to the unpopular

[3] WatchTower's plugin logic for the mobile browser is implemented in the Cumulus caching daemon (§3) since Chrome for Android does not support extensions.




(a) 4G LTE cellular network. (b) Residential WiFi network. (c) Wired broadband network.

**Figure 11: Speedups for WatchTower and always-on proxying; the performance baseline was a browser that never used proxying. Note that speedups less than 1 represent an *increase* in page load times.**

sites that the user visits. Thus, proxies will measure latencies to all popular origins, and to the unpopular origins that the proxies' clients actually care about. The client-side plugin is responsible for capturing dependency graphs for long-tail pages, as well as measuring the associated client-origin latencies.

Web browsing patterns of individual users are stable over time [51]. The dependency graph for an individual page also changes slowly—the overall graph shape is stable over multiple days, and the origin associated with each vertex's URL rarely changes, even though the full URL often does [16]. So, capturing a page's dependency graph once every few days is sufficient.

WatchTower's design is agnostic to the latency measurement scheme used by proxies and clients. Our prototype uses weighted averages of simple pings to measure RTTs, but could leverage prior work which describes how to efficiently perform latency estimates that scale to millions of hosts [19, 30, 54, 65].

## 6.3 Evaluation

To evaluate WatchTower, we first compared its performance to that of an always-on proxying system. We placed Cumulus proxies on the EC2 nodes described in §3. With WatchTower, each page load used the proxy that was predicted to give the best load time; if all proxies were predicted to hurt load times relative to a no-proxy scenario, WatchTower disabled proxying. In the always-on proxying system, all page loads used the proxy that was geographically closest to the client.[4] Since the client location was fixed to Massachusetts, the always-on system consistently used the proxy in Massachusetts. To simplify the experimental setup, WatchTower collected dependency graphs and RTT values immediately prior to each experiment. At the end of this section, we examine WatchTower's sensitivity to fluctuating RTTs and stale dependency graphs. Unless otherwise stated, experiments used cold browser caches.

Figure 11 compares the speedups achieved by WatchTower and always-on proxying, relative to a browser that never used proxying. The experiment considered HTTPS-sharding and HTTP-only resolution which, unlike full RDR, respect end-to-end HTTPS security. Results used the Alexa top 500 sites. As shown, WatchTower significantly reduces the probability that, when proxies are used, they hurt load times. In the residential WiFi network, this probability drops from 23.3%–37.8% with the always-on model, to 0.64%–0.97% with

[4]This represents the proxy selection heuristic for existing proxies like Flywheel [2] and Opera Mini [52].

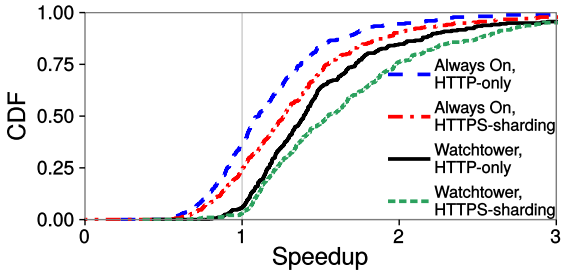



**Figure 12: Speedups for WatchTower and always-on proxying with a cellular network. In contrast to Figure 11(a), performance is defined using Speed Index [26] instead of the traditional page load time definition.**

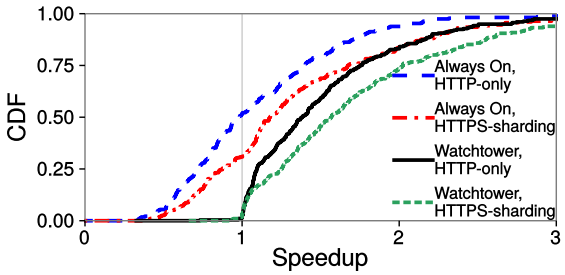



**Figure 13: Speedups for WatchTower and always-on proxying with warm browser caches on a cellular network. We saw similar trends for the other networks.**

WatchTower. The reduction is even larger in the wired broadband network, dropping from 49.1%–62.6% to 0.61%–1.23%. The reason is that, when $RTT_{client-origin}$ values approach $RTT_{client-proxy}$, WatchTower has more opportunities to disable proxying and avoid inefficient network detours for object fetches.

Using selective proxying, WatchTower outperforms the always-on model for all of the networks and both HTTP-only and HTTPS-sharding resolution. For example, on a cellular network, median speedups for always-on proxying were 1.14× and 1.45× for HTTP-only and HTTPS-sharding resolution, respectively. With WatchTower, speedups were 1.51× and 1.87×. In terms of raw savings, this equated to removing 1.22 seconds–1.85 seconds of load time relative to the always-on model. Table 2 quantifies these raw savings, which are quite impressive, since web developers are elated to shave *tens of milliseconds* from load times [11, 12, 24].

| Client network | WatchTower HTTP-only | WatchTower HTTPS-sharding | Always-On HTTP-only | Always-On HTTPS-sharding |
|---|---|---|---|---|
| 4G LTE Cellular | 1.39× | 1.54× | 1.10× | 1.26× |
| Residential WiFi | 1.18× | 1.41× | 1.06× | 1.15× |
| Wired Broadband | 1.07× | 1.19× | 1.02× | 1.08× |

**Table 3: Median speedups in Speed Index for WatchTower and always-on proxying; the performance baseline is a browser that never used proxying.**

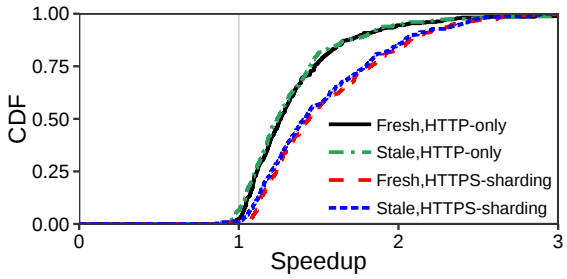



**Figure 14: Speedups for WatchTower using fresh and 24-hours-stale latency measurements on a WiFi network. We saw similar trends on the other networks.**

| Approach | PLT (sec) |
|---|---|
| WatchTower (co-located, HTTPS-sharding) | 2.32 |
| WatchTower (co-located, HTTP-only) | 3.42 |
| WatchTower (cloud, HTTPS-sharding) | 2.47 |
| WatchTower (cloud, HTTP-only) | 3.50 |
| Vroom-style push+preload hints | 3.53 |
| Vanilla HTTP/2 push | 4.05 |
| Default | 5.18 |

**Table 4: Push+preload versus RDR: median page load times (PLTs) in the cellular network (with a Nexus 5 phone). With co-located proxying, each origin server had a 1 ms RTT to a co-located, per-origin proxy (§4); the cloud proxy setting used the proxies described in §3.**

**Speed Index:** The traditional definition for "page load time" states that a page is not loaded until a browser has fetched all of the external resources (e.g., images and JavaScript files) that are referenced by the page's top-level HTML. In contrast, a page's Speed Index [26] represents the time that a browser needs to completely render the pixels in the initial view of the page. By focusing on the visual progression of "above-the-fold" content, Speed Index tries to capture the subjective preference of a human who favors a quickly-rendering page, even if some of the JavaScript or below-the-fold content is not immediately ready. Speed Index cannot represent the fact that a fully-rendered page is not truly ready until JavaScript event handlers have been registered, and JavaScript timers have started to implement animations. Thus, Speed Index is not strictly better than the traditional PLT metric [46]. Nevertheless, Speed Index provides a useful second perspective for understanding page loads.

Table 3 shows that, across all three test networks, WatchTower provides significant reductions in Speed Index compared to an Always-On proxying system. Figure 12 illustrates those savings for the cellular network scenario, where median speedups for always-on proxying were 1.10× and 1.26× for HTTP-only and HTTPS-sharding resolution, respectively. With WatchTower, speedups were 1.39× and 1.54×. In terms of raw savings, this equated to removing 428–562 milliseconds of load time relative to the always-on model.

These savings, while practically useful, are less than the savings that WatchTower provides for the traditional definition of page load time. This discrepancy is expected, since WatchTower's algorithms (§6.1) specifically target the traditional load time metric, not Speed Index. Modifying WatchTower's algorithms to target Speed Index is an important area for future research.

**Preloading and Server Push:** HTTP/2 [9] allows a web server to proactively send objects to clients. At first glance, server push might seem to unlock many of WatchTower's benefits, since a server can stream objects to a client without waiting for explicit requests. Using Mahimahi [48], we experimentally compared WatchTower to a proxy-less scenario in which servers pushed all static objects. We also evaluated a Vroom-style push+preload system [58] in which each frame in a page pushed objects from the frame's origin, and used preload hints [67] to encourage the browser to fetch static objects from other origins ahead of time.

We ran the experiments on all networks, but due to space restrictions, we only show results for the cellular network (Table 4). There are four key observations:

- WatchTower's performance is slightly better with proxy co-location. However, just five cloud-based proxy choices are sufficient to unlock roughly 95% of the benefits enabled by co-located proxies.
- WatchTower with HTTPS-sharding or HTTP-only is 13.5%–42.7% faster than vanilla server push. One reason is that, in a standard push approach, the server can only push statically-referenced page objects, not dynamic ones generated by (say) `XMLHttpRequests`. Further, a server may accidentally push objects that are not on the dynamic critical path [44, 70]; these pushes contend for last-mile bandwidth that is coveted by critical path objects whose fetches are *currently blocking* the client-side page load. In contrast, WatchTower's RDR approach handles both static and dynamic objects, and streams objects to the client in the order that the client will naturally request them.
- A Vroom-style approach is only 0.8%–3.1% slower than WatchTower in HTTP-only mode. This is because 68% of the objects in the test corpus used HTTPS; Vroom can push and preload HTTPS objects, whereas WatchTower in HTTP-only mode cannot optimize HTTPS traffic. WatchTower in HTTP-only mode is still competitive because (unlike Vroom) it can resolve dynamically-generated URLs, and avoid blocking issues involving non-critical-path objects.
- WatchTower with HTTPS-sharding is 30.1%–34.3% faster than Vroom. In this mode, WatchTower leverages full URL coverage and optimal object transmission scheduling for both HTTP and HTTPS traffic.

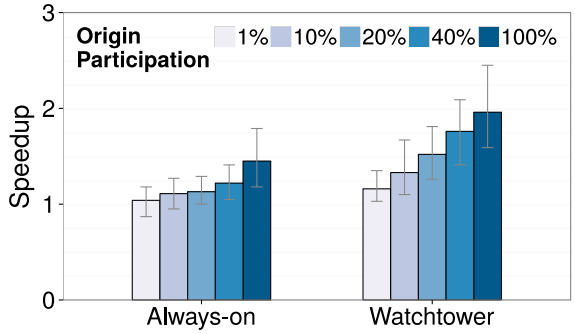



**Figure 15: WatchTower and always-on proxying when the most popular $X$% of origins deployed proxies. These results used HTTPS sharding and a cellular network.**

**Partial deployment:** In the real world, some origins will not deploy proxies. Figure 15 shows the benefits of incremental proxy deployment. The experiments used a random set of 100 pages from the Alexa top 10,000; in each batch of experiments, only the most popular $X$% of origins deployed proxies, where popularity of an origin was defined by the number of references to that origin in the Alexa top 500 pages. As expected, Figure 15 shows that proxying (either always-on or WatchTower) provides more benefits when more domains run proxies. However, WatchTower's benefits improve faster than those provided by always-on proxying; furthermore, at every level of origin participation, WatchTower provided more benefits than always-on proxying. Also note that, for $X$=20% and $X$=40%, WatchTower provided 79% and 89% of the speedups achieved when $X$=100%.

We also ran an experiment in which only the top-level origin for each page deployed a proxy. For cellular networks and HTTPS sharding, WatchTower provided a 1.65× median improvement, compared to 1.87× when all origins deployed proxies. Always-on, top-level-only proxying enabled a 1.17× improvement, compared to 1.45× when all origins deployed proxies.

**Personalized Pages:** As described in §6.2, WatchTower collects dependency graphs using two mechanisms: proxies collect graphs for popular sites, while client-side plugins collect graphs for long-tail pages. Since the plugins supply cookies during the fetching of page content, the plugins are able to construct *personalized* dependency graphs that reflect user-specific page content. In contrast, WatchTower proxies do not supply cookies when building dependency graphs for popular sites; as a result, those graphs represent non-personalized pages. So, for popular sites, WatchTower may use a non-personalized graph to determine how a client should load a personalized page.

For pages like `facebook.com`, there is a large difference between two personalized versions (e.g., for separate users); deltas are also large between a personalized version and a non-personalized version (i.e., the login page). Fortunately, for many other sites, the differences between page versions are small, and do not impact WatchTower's accuracy. For example, we found 20 random sites in the Alexa top 500 that supported user accounts, but were not Facebook-style social networks. For each site, we made a user account, loaded a personalized page on a mobile phone over a cellular network, and recorded the median load time with and without proxy assistance. Finally, we fed the *non-personalized* dependency graph to WatchTower, and asked whether a proxy would decrease load times. For all 20 sites, WatchTower made the correct prediction when deciding between HTTPS-sharding proxying and no proxying. WatchTower also had perfect accuracy for HTTP-only proxying. In other words, for all 20 sites, non-personalized graphs were similar enough to personalized graphs that WatchTower's accuracy was unaffected.

**Energy usage:** WatchTower reduces load times, but does require extra client-side computation. This extra computation requires additional power, which is potentially concerning on battery-constrained mobile devices. To examine WatchTower's impact on energy usage, we connected a Nexus 5 Android phone to a Monsoon power monitor [40]; the phone had a 2.7 GHz quad core processor with 3 GB of RAM. On a cellular network, WatchTower using HTTPS sharding reduced power consumption by a median of 22%. Using HTTP-only resolution, WatchTower reduced energy usage by a median of 11%. In both cases, the extra energy consumed by WatchTower computation was far offset by the energy saved by loading pages faster and allowing the radio to be powered off sooner.

**Browser caching:** As discussed in Section 4, remote dependency resolution can reduce load times even when browser caches are warm. However, warm caches reduce the number of object fetches that a proxy can optimize. To determine whether WatchTower provides benefits in these scenarios, we reran the caching experiment from §4, comparing WatchTower and an always-on proxy to a browser that never used a proxy. WatchTower used the analytic models from §6.1, associating each cached object with an estimated fetch latency of 0 ms. Figure 13 shows that WatchTower still provides significant benefits if caches are warm. For the HTTP-only scheme, an always-on proxy slows 51.4% of page loads; for WatchTower, only 1.8% of page loads suffer. For HTTPS-sharding, always-on proxying hurts 30.8% of loads; WatchTower reduces that number to 1.7%. For both HTTPS-sharding and HTTP-only resolution, WatchTower reduces median load times by roughly 40%.

**Inaccurate measurements:** WatchTower uses empirical measurements to estimate the network latencies between clients, proxies, and origin servers. However, real network conditions are not stationary, so WatchTower's latency estimates may differ from the actual latencies at page load time. Figure 14 shows that WatchTower's benefits are largely insensitive to the use of latency estimates that are 24 hours stale. The probability that WatchTower hurts load times increases from 0.97% to 2.4%; median load times are within 0.59%–1.35% of those which use fresh measurements. WatchTower's benefits are largely unchanged due to the wide performance gaps between the proxy schemes from which WatchTower must select—variance in network latency is rarely large enough to change the best scheme to use in a particular situation.

We also used the Mahimahi web replay framework [48] to inject controlled amounts of estimation error. We modeled error as a Gaussian process with a mean of -3% and various amounts of standard deviation; the slightly-less-than-zero mean was inspired by observations of the King latency estimation algorithm which slightly underestimates true RTTs [30]. We tested WatchTower's performance on a cellular network, with HTTPS sharding enabled.

When WatchTower had perfectly accurate RTT estimates, WatchTower provided a median speedup of 1.79×. With Gaussian estimation error that had a mean of -3% and a standard deviation of 10%, WatchTower provided a median speedup of 1.73×. If the standard deviation doubled to 20%, WatchTower's median speedup only decreased to 1.65×.

Dependency graphs are stable over days [16], and we found that WatchTower is insensitive to the use of stale graphs to drive predictions. For example, in the cellular network, WatchTower's benefit across all RDR schemes drops by less than 3% with 24-hour-stale graphs.

**WatchTower overheads:** Having a WatchTower proxy and client measure latencies to the 4057 distinct origins in our 500 site corpus required 1.95 MB of ping traffic each; shipping proxy-origin estimates to a client used 28.9 KB. A WatchTower proxy must also send dependency graphs to clients. Generating those graphs merely requires loading a page and inspecting the "Initiator" fields in the browser's debugging output [28]. The median (95th percentile) gzipped dependency graph sizes in our corpus were 2.9 KB (92.7 KB). In total, all 500 dependency graphs were 1.71 MB, which is smaller than a typical web page [33]. At page load time, the client plugin must perform the graph analysis from §6.1, which takes at worst 3 ms per page load.

WatchTower proxies mimic client-generated HTTP headers to ensure that RDR page loads fetch the same objects that would be fetched by traditional (i.e., proxy-less) page loads (§3). Nevertheless, a WatchTower proxy may push unnecessary resources to a client. For example, if a URL is created by JavaScript code and embeds the current time, then a proxy and its client may generate different versions of the URL. Such discrepancies will not affect the *correctness* of a page load—the client's request will simply miss in WatchTower's client-side cache and require an additional round trip to fetch the appropriate object. However, the pushing of unnecessary content does impose a bandwidth penalty. Across the Alexa top 500 sites, the median page load fetched 31.3 KB of unnecessary objects; at the 95th percentile, the wastage was 214 KB. To put those numbers in perspective, the size of the median test page was 1.51 MB.

## 7 ADDITIONAL RELATED WORK

**Mobile web optimizations:** Klotski [16] uses offline page analyses to identify high-priority objects (according to a user-specific utility function), which are pushed to clients using a proxy. Thus, Klotski can only securely accelerate loads for HTTP pages, and like simpler push policies, is limited to pushing static objects that are discovered offline. In contrast, WatchTower can securely speed up the loading of static and dynamic objects that are served with HTTP or HTTPS.

AMP [25] requires mobile pages to be rewritten using restricted forms of HTML, JavaScript, and CSS, that are known to be performant; compliant pages are served from Google CDNs. Unlike AMP, WatchTower is able to accelerate loads of legacy pages. Moreover, WatchTower can speed up AMP page loads by hiding serial network round trips; these latencies are significant even for content served from CDNs as individual requests must traverse a client's slow last mile.

Polaris [44] uses a client-side scheduler to reorder requests according to pre-computed dependencies between a page's objects; the goal is to maximally overlap round trips over the client's last mile. Polaris naturally preserves end-to-end security (by not using a proxy), but all round trips are still incurred over the client's slow last-mile link, limiting potential benefits. Instead, WatchTower selectively shifts round trips to fast proxy-origin links, further reducing last-mile round trips. WatchTower's proxy could speed up its page loads by loading Polaris-optimized pages.

**Secure middleboxes:** BlindBox [59] enables a proxy to perform deep packet inspection on an HTTPS flow, only discovering cleartext that matches a preset filter. This is insufficient for remote dependency resolution, since an RDR proxy must evaluate entire cleartext objects, not parts of them that match a filter. mcTLS [42] is more flexible, allowing a user and a particular HTTPS origin to authenticate a middlebox, and selectively disclose data using different encryption streams. mcTLS can allow a middlebox to see and evaluate an entire cleartext object. At first glance, this might appear sufficient to build a trusted RDR proxy for HTTPS content. However, when a client initiates a page load, she lacks a priori knowledge of the HTTPS origins that will contribute objects to the page. Further, a typical client loads many different pages in a day. Thus, RDR proxies using mcTLS would face a challenging key management problem—users would have to make frequent pairwise trust decisions involving many origins. HTTPS sharding imposes a smaller cognitive burden on users, and does not require HTTPS content to be shared with anyone except the HTTPS endpoints.

## 8 CONCLUSION

In this paper, we identify and eliminate two key barriers to efficient remote dependency resolution. First, we explain why HTTPS introduces a tension between security and performance with remote dependency resolution; we resolve this tension using HTTPS sharding, a new resolution technique which allows HTTPS traffic to be proxied without revealing sensitive data to untrusted origins. Second, we demonstrate that always-on proxying (the default mode for state-of-the-art proxies) actually increases load times in many cases. We introduce WatchTower, a new proxy which uses HTTPS sharding and selective proxying to load pages 21.2%–41.3% faster than state-of-the-art proxies and server push systems, while preserving the end-to-end security of HTTPS.

## ACKNOWLEDGMENTS

We thank Haitham Hassanieh, Mohammad Alizadeh, the anonymous reviewers, and our shepherd Sharad Agarwal for their helpful comments and suggestions. This research was partially supported by NSF grant CNS-1407470.

## REFERENCES

[1] http://www.w3.org/Protocols/rfc2616/rfc2616.html.
[2] 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.
[3] Alexa. Top Sites in the United States. http://www.alexa.com/topsites/countries/US, 2018.
[4] Amazon. Amazon Cloudfront – Content Delivery Network (CDN). https://aws.amazon.com/cloudfront/, 2016.
[5] Amazon. Silk Web Browser. https://docs.aws.amazon.com/silk/latest/developerguide/introduction.html, 2018.
[6] Apache Software Foundation. Reverse Proxy Guide. https://httpd.apache.org/docs/2.4/howto/reverse_proxy.html, 2018.
[7] A. Barth. HTTP State Management Mechanism. https://tools.ietf.org/html/rfc6265, April 2011.

[8] M. Belshe. More Bandwidth Doesn't Matter (Much). https://goo.gl/PFDGMi, April 8, 2010.

[9] M. Belshe, R. Peon, and M. Thomson. HTTP/2.0 Draft Specifications. https://http2.github.io/, 2018.

[10] D. Bhattacherjee, M. Tirmazi, and A. Singla. A Cloud-based Content Gathering Network. In *Proceedings of HotCloud*, 2017.

[11] 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.

[12] 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.

[13] Bro Project. The Bro Network Security Monitor. https://www.bro.org/, 2018.

[14] M. Brown. Destination NAT with netfilter (DNAT). http://linux-ip.net/html/nat-dnat.html, March 14, 2007.

[15] 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*, IMC '11. ACM, 2011.

[16] 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.

[17] F. Callegati, W. Cerroni, and M. Ramilli. Man-in-the-Middle Attack to the HTTPS Protocol. *IEEE Security & Privacy*, 7(1), 2009.

[18] Cisco. Snort: Network Intrusion Detection and Prevention System. https://www.snort.org/, 2018.

[19] F. Dabek, R. Cox, M. F. Kaashoek, and R. Morris. Vivaldi: A Decentralized Network Coordinate System. In *Proceedings of SIGCOMM*, 2004.

[20] E. Enge. Mobile vs Desktop Usage in 2018: Mobile takes the lead. https://www.stonetemple.com/mobile-vs-desktop-usage-study/.

[21] 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.

[22] D. Fisher. HTTP Caching FAQ. https://developer.mozilla.org/en-US/docs/Web/HTTP/Caching_FAQ.

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

[24] 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.

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

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

[27] Google. WebP: A new image format for the Web. https://developers.google.com/speed/webp/?hl=en, March 4, 2016.

[28] Google. Chrome Dev Tools: Network Analysis Reference. https://developers.google.com/web/tools/chrome-devtools/network-performance/reference#analyze, May 16, 2017.

[29] Google. Transparency Report: HTTPS encryption on the web. https://transparencyreport.google.com/https/overview?hl=en, October, 2018.

[30] K. Gummadi, S. Saroiu, and S. Gribble. King: Estimating Latency Between Arbitrary Internet End-host. In *Proceedings of the 2002 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM Workshop on Internet Measurement. ACM, 2002.

[31] HAProxy. HAProxy: The Reliable, High Performance TCP/HTTP Load Balancer. http://www.haproxy.org/, 2018.

[32] A. Hidayat. PhantomJS. http://phantomjs.org/, 2018.

[33] HTTP Archive. Interesting stats. http://httparchive.org/interesting.php#bytesTotal, June 14, 2017.

[34] L. S. Huang, A. Rice, E. Ellingsen, and C. Jackson. Analyzing Forged SSL Certificates in the Wild. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, SP. IEEE Computer Society, 2014.

[35] JetBlue. Fly-Fi. http://www.jetblue.com/flying-on-jetblue/wifi/, 2018.

[36] G. Lea. Caldera spin-off pushes Linux thin clients. The Register. http://www.theregister.co.uk/1999/04/27/caldera_spinoff_pushes_linux_thin/, April 27, 1999.

[37] T. Mather. geoip-lookup(1): Linux man page. http://linux.die.net/man/1/geoip-lookup, 2018.

[38] B. McQuade, D. Phan, and M. Vajolahi. Instant Mobile Websites: Techniques and Best Practices. https://www.youtube.com/watch?v=Bzw8-ZLpwtw, May 16, 2013.

[39] D. Meyer. Nokia: Yes, we decrypt your HTTPS data, but don't worry about it. GigaOM. https://gigaom.com/2013/01/10/nokia-yes-we-decrypt-your-https-data-but-dont-worry-about-it/, January 10, 2013.

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

[41] D. Naylor, A. Finamore, I. Leontiadis, Y. Grunenberger, M. Mellia, M. Munafò, K. Papagiannaki, and P. Steenkiste. The Cost of the "S" in HTTPS. In *Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies*, CoNEXT. ACM, 2014.

[42] D. Naylor, K. Schomp, M. Varvello, I. Leontiadis, J. Blackburn, D. R. López, K. Papagiannaki, P. Rodriguez Rodriguez, and P. Steenkiste. Multi-Context TLS (mcTLS): Enabling Secure In-Network Functionality in TLS. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM. ACM, 2015.

[43] J. Nejati and A. Balasubramanian. An In-depth Study of Mobile Browser Performance. In *Proceedings of WWW*, 2016.

[44] 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.

[45] 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.

[46] 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.

[47] R. Netravali, A. Sivaraman, J. Mickens, and H. Balakrishnan. Supplementary Material For WatchTower, 2018. http://web.cs.ucla.edu/~ravi/publications/watchtower_supplementary_material.pdf.

[48] 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.

[49] NGINX Inc. High-performance Load Balancer, Web Server, and Reverse Proxy. https://www.nginx.com/, 2018.

[50] 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.

[51] L. Olejnik, C. Castelluccia, and A. Janc. Why Johnny Can't Browse in Peace: On the Uniqueness of Web Browsing History Patterns. In *Privacy Enhancing Technologies Symposium*, HotPETs, 2012.

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

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

[54] M. Pias, J. Crowcroft, S. Wilbur, T. Harris, and S. Bhatti. Lighthouse for Scalable Distributed Location. In *Proceedings of IPTPS*, 2003.

[55] Progress Software Corporation. Telerik fiddler. http://www.telerik.com/fiddler, 2018.

[56] E. Rescorla. HTTP Over TLS. https://tools.ietf.org/html/rfc2818, May 2000.

[57] S. Rosen, B. Han, S. Hao, Z. M. Mao, and F. Qian. Push or Request: An Investigation of HTTP/2 Server Push for Improving Mobile Performance. In *Proceedings of the 26th International Conference on World Wide Web*, WWW. International World Wide Web Conferences Steering Committee, 2017.

[58] 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.

[59] J. Sherry, C. Lan, R. A. Popa, and S. Ratnasamy. BlindBox: Deep Packet Inspection over Encrypted Traffic. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM. ACM, 2015.

[60] 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.

[61] 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.

[62] 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 International on Conference on Emerging Networking Experiments and Technologies*, CoNEXT '14, pages 325–336, New York, NY, USA, 2014. ACM.

[63] G. Sterling. Mobile Devices Now Driving 56 Percent Of Traffic To Top Sites. https://marketingland.com/mobile-top-sites-165725, 2016.

[64] S. Sundaresan, N. Feamster, R. Teixeira, and N. Magharei. Measuring and Mitigating Web Performance Bottlenecks in Broadband Access Networks. In *Proceedings of the 2013 Conference on Internet Measurement Conference*, IMC, New York, NY, USA, 2013. ACM.

[65] M. Szymaniak, D. Presotto, G. Pierre, and M. van Steen. Practical Large-Scale Latency Estimation. *Computer Networks*, 52:1343–1364, 2008.

[66] J. Volpe. Nokia Xpress brings cloud-based compression to the Lumia line. Engadget. https://www.engadget.com/2012/10/03/nokia-xpress-brings-cloud-based-compression-to-the-lumia-line/, October 3, 2012.

[67] W3C. Preload. Editor's Draft. https://w3c.github.io/preload/, January 9, 2018.

[68] L. Wang, K. Park, R. Pang, V. S. Pai, and L. L. Peterson. Reliability and security in the codeen content distribution network. ATC, 2004.

[69] 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.

[70] 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.

[71] 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.

[72] Z. Wang, F. X. Lin, L. Zhong, and M. Chishtie. Why Are Web Browsers Slow on Smartphones? In *Proceedings of the 12th Workshop on Mobile Computing Systems and Applications*, HotMobile. ACM, 2011.

[73] D. Wessels, H. Nordstrom, A. Jeffries, A. Rousskov, F. Chemolli, R. Collins, and G. Serassio. Squid: Optimising Web Delivery. http://www.squid-cache.org/, 2018.

[74] T. Zimmermann, B. Wolters, O. Hohlfeld, and K. Wehrle. Is the Web ready for HTTP/2 Server Push? In *Proceedings of the 14th ACM International on Conference on Emerging Networking Experiments and Technologies*, CoNEXT. ACM, 2018.