Supplementary Material For WatchTower

A COOKIES, PROXYING, AND USER PRIVACY

Web sites use cookies to store client-side, per-user metadata like configuration preferences. When a browser generates a request for `http://foo.com/someObj`, the browser attaches any client-side cookies that belong to `foo.com`, using HTTP headers in the request to transmit the cookie values to the web server. If a cookie value is marked as `Secure` [38], the browser will only transmit the value to `foo.com`'s HTTPS servers (meaning that requests for `foo.com`'s HTTP objects will not include `Secure` cookie values).

Even non-`Secure` cookie values may contain sensitive information, because cookies are frequently used by web sites to track user browsing habits [69]. However, if users do not share their cookies with proxies, those proxies will be unable to fetch customized data on behalf of the users. Thus, Cumulus and all other proxy systems have to make a policy decision about which cookies a user will share with proxies. For example, Flywheel [2] only handles HTTP traffic, so Flywheel proxies only see non-`Secure` cookies. In contrast, Opera Mini [56] processes both HTTP and HTTPS traffic, and sees all user cookies.

Cumulus defines two different policies for handling cookies. When using the *permissive cookie-sharing policy*, Cumulus does the following:

- For HTTP-only proxying, the client-side Cumulus agent sends all non-`Secure` HTTP cookies to the remote proxy. This privacy model is equivalent to that of Flywheel.
- In full dependency resolution mode, Cumulus sends both `Secure` and non-`Secure` cookies to the remote proxy. The resulting privacy model is similar to that of Opera Mini.
- When performing HTTPS-sharding (§4), Cumulus sends all non-`Secure` cookies to the proxy for HTTPS origin `foo.com`, but only the `Secure` cookies that belong to `foo.com`. This approach never exposes `Secure` cookies outside their origin, while still allowing HTTPS proxying of customized content. The scope of disclosure for non-`Secure` cookies is the same as in Flywheel.

As an alternative to the permissive sharing policy, Cumulus also supports a *restricted sharing* policy. In this policy, the client-side Cumulus agent performs no bulk synchronization of cookies with the proxy. Thus, the only cookies that a proxy can see are the ones that are embedded in a client's initial `RESOLVE` request for top-level HTML (see Figures 2 and 4). Importantly, *neither cookie policy can break the correctness of the user-visible page load.* The reason is that Cumulus' client-side caching agent only defines a request to hit in the cache if all of the local browser's request headers match the ones that were generated by the remote proxy. So, if a remote proxy fetched `http://foo.com/obj` without sending any cookies, but the request from the user's browser *does* include cookies, the fetch from the user's browser will not hit in the Cumulus cache, and the user's browser will directly fetch the object from foo.com. All of our evaluation results use the restrictive sharing policy; thus, our performance numbers represent conservative estimates of a proxy's ability to reduce load times.

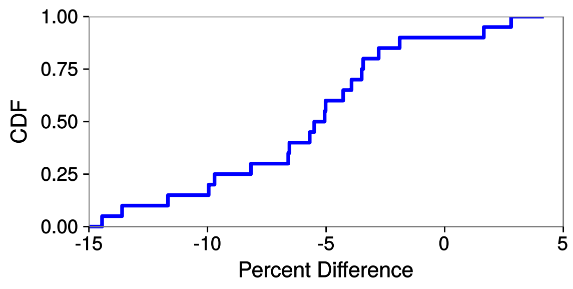

Figure 16: Distribution of the percentage difference in page load times between Opera Mini and Cumulus running in full dependency resolution mode.

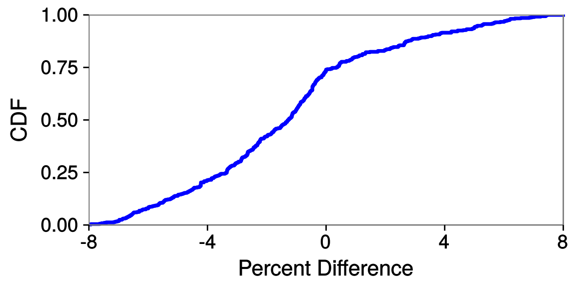

Figure 17: Distribution of the percent difference in page load times between Flywheel and compression-only Cumulus.

B CALIBRATING CUMULUS

To calibrate the performance of compression-only Cumulus with respect to Flywheel, we loaded the Alexa US Top 500 pages with the Chrome browser (version 51). We configured the browser to use either the Cumulus proxy, or the Flywheel proxy. The location of the Flywheel proxy (as chosen by the Chrome Proxy Plugin [27]) was northern California; to provide a fair comparison with Cumulus, we ran the Cumulus proxy on an Amazon EC2 instance in northern California. We loaded each page five times using Cumulus, and five times using Flywheel; we took the average load time using both systems, and then recorded the percent difference in average load time. Figure 17 shows the CDF of that difference, indicating that the median Cumulus load time is 1.27% faster than the corresponding Flywheel load time. Thus, we believe that Cumulus in compression-only mode is competitive with a state-of-the-art compression proxy, at least with respect to page load time (which is the focus of this paper).

We also calibrated the performance of Cumulus' remote dependency resolution with respect to Opera Mini [13, 56], an existing proxy that performs full dependency resolution. Opera Mini defaults to requesting the mobile versions of web pages, so our test corpus used a set of 20 mobile pages from the Alexa US Top 500. The client was a desktop browser. In the Cumulus experiments, the client set its `User-agent` HTTP header to a value which convinced web servers to return the mobile versions of pages. The location of Opera Mini's proxy was in Virginia, so we ran the Cumulus proxy on an Amazon EC2 instance in Virginia.

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

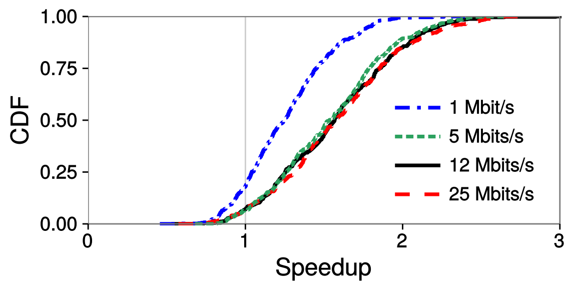

Figure 18: For a last-mile RTT of 50 ms, speedup is mostly insensitive to last-mile bandwidth. These experiments used full dependency resolution, but the trends are similar for HTTP-only and HTTPS-sharding.

Figure 16 compares load times using Opera Mini and Cumulus running full RDR. Cumulus exhibited similar performance to Opera Mini, providing a median load time that was 5.3% faster. Thus, we believe that Cumulus is representative of state-of-the-art proxies that perform remote dependency resolution.

C REMOTE DEPENDENCY RESOLUTION AND LAST-MILE BANDWIDTH

In this section, we provide a simple but illustrative demonstration of how the effectiveness of remote dependency resolution is generally insensitive to link bandwidth. These results, when combined with those from Section 5.1, validate prior research which indicates that page loads are much more sensitive to latency than bandwidth [2, 8, 42, 68].

Using Mahimahi, we created an emulated network in which:

- the proxy had an infinite bandwidth, zero latency link to origin servers,
- $RTT_{client-proxy}$ and $RTT_{client-origin}$ were fixed at 50 ms, and
- last-mile bandwidth was set to 1, 5, 12, or 25 Mbits/s.

Figure 18 shows that, across all 500 pages in our test corpus, speedups from full dependency resolution were largely insensitive to last-mile bandwidth. Since most web objects are small [24], an object's bandwidth-induced transfer delay is generally dwarfed by its latency-induced delay. For example, consider the Google page (which was 63 KB in size) and the TMZ page (which was 6.9 MB in size). Proxy speedups for the Google page varied less than 1.7% across all link rates. In contrast, for the TMZ page, the speedup was 1.64× at 1 Mbit/s, but 1.88× at 5 Mbit/s and above. For a larger page like TMZ, the benefits from remote dependency resolution initially increase as bandwidth increases (and latency becomes a more dominant factor in overall fetch costs). However, the benefits eventually plateau as non-parallelizable fetch latencies dominate the overall page load time.

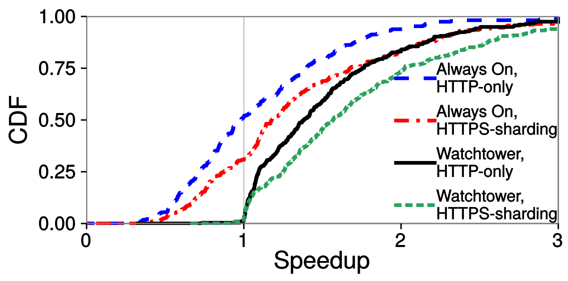

Figure 19: Speedups for WatchTower and always-on proxying with warm browser caches on a cellular network. We saw similar trends for the other networks.

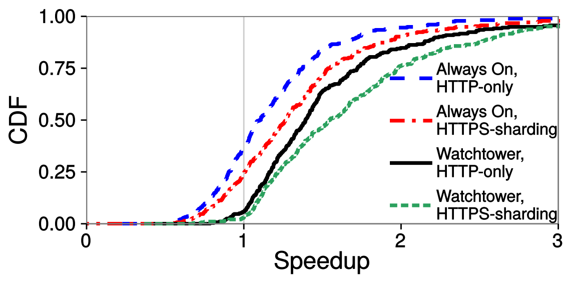

Figure 20: Speedups for WatchTower and always-on proxying with a cellular network. In contrast to Figure 11(a), performance is defined using Speed Index [29] instead of the traditional page load time definition.

D EVALUATING WATCHTOWER WITH WARM BROWSER CACHES

Warm caches reduce the number of object fetches that a proxy can optimize. To determine whether Watchtower provides benefits in these situations, we reran the caching experiment from Section 4, comparing Watchtower and an always-on proxy to the performance baseline of a browser that never used a proxy. Watchtower employed the same analytic models from Section 6.1, associating each cached object with an estimated fetch latency of 0 ms.

Figure 19 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, an always-on approach hurts 30.8% of loads; Watchtower reduces that number to 1.7%. For both HTTPS-sharding and HTTP-only, Watchtower reduces median load times by roughly 40%.

E EVALUATING WATCHTOWER ON 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 [29] 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. Nevertheless, Speed Index provides a useful second perspective for understanding page loads.

Table 5 shows that, across all three test networks, WatchTower provides significant reductions in Speed Index compared to an Always-On proxying system. Figure 20 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.