Oblique: Accelerating Page Loads Using Symbolic Execution

Ronny Ko, James Mickens
Harvard University

Blake Loring
Royal Holloway, University of London

Ravi Netravali
UCLA

Abstract

Mobile devices are often stuck behind high-latency links. Unfortunately for mobile browsers, latency (not bandwidth) is often the key influence on page load time. Proxy-based web accelerators hide last-mile latency by analyzing a page's content, and informing clients about useful objects to prefetch. However, most accelerators require content providers to divulge cleartext HTTPS data to third-party analysis servers. Acceleration systems can be installed on first-party web servers, avoiding the violation of end-to-end TLS security; however, due to the administrative overhead (and additional VM costs) associated with running an accelerator, many first-party content providers would prefer to outsource the acceleration work—if outsourcing could be secure.

In this paper, we introduce Oblique, a third-party web accelerator which enables secure outsourcing of page analysis. Oblique symbolically executes the client-side of a page load, generating a prefetch list of symbolic URLs. Each symbolic URL describes a URL that a client browser should fetch, given user-specific values for cookies, the `User-Agent` string, and other sensitive variables. Those sensitive values are never revealed to Oblique's analysis server. Instead, during a real page load, the user's browser concretizes URLs by reading sensitive local state; the browser can then prefetch the associated objects. Experiments involving real sites demonstrate that Oblique preserves TLS integrity while providing faster page loads than state-of-the-art accelerators. For popular sites, Oblique is also financially cheaper in terms of VM costs.

1 Introduction

Two trends are reshaping modern web services: the increasing prevalence of mobile traffic, and the continued shift from HTTP to HTTPS. 53% of all page requests now originate from smartphones [5]. 90% of those requests use HTTPS [16].

Many mobile users (particularly in emerging markets) are still stuck behind slow 3G and 4G links; even high-bandwidth 5G links often suffer from 4G latencies [11]. Unfortunately, page load times are usually determined by latency, not bandwidth [21, 24]. A variety of mobile page accelerators try to mask last-mile latency by (1) analyzing the objects (e.g., HTML and JavaScript files) that are contained by a page, and then (2) reducing the perceived fetch latencies for those objects (e.g., using server-side pushing [1,24,29,34,35] or client-side prefetching [21,29]). Unfortunately, the shift from HTTP to HTTPS has created tensions between security, performance, and the financial cost of hosting a web site. Accelerators like Silk [1] that perform remote dependency resolution [3,24,35] route client traffic through third-party proxies; these proxies are owned by browser vendors or mobile providers, and are operated for the benefit of customers. The proxies require access to cleartext HTTPS content to determine which objects to prefetch (§2). Thus, content providers that use HTTPS are faced with a dilemma: allow third-party proxies to man-in-the-middle TLS connections, or forgo the performance benefits provided by outsourced web accelerators. The former choice breaks end-to-end TLS security, and the latter option hurts page load times.

A content provider could decide to run a first-party web accelerator like Vroom [29] locally; this approach would avoid revealing cleartext HTTPS data to an untrusted middlebox. However, the content provider would incur a new financial penalty. Running a web accelerator requires extra CPU cycles and memory space beyond what is required to run a traditional web server. The content provider would have to pay for those extra VM resources.

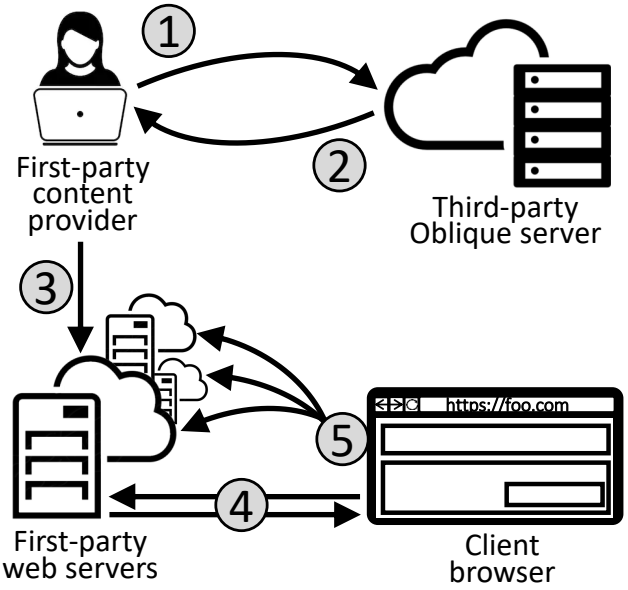


Figure 1: Overview of Oblique's design. A developer uploads page content to Oblique's analysis server (①). Oblique returns the path constraint tree for the page (②). The developer uploads the page content to web servers (③), injecting Oblique's JavaScript library into the page's HTML. Later, when a user loads the page (④), the prefetching library uses the path constraint tree to prefetch objects (⑤).

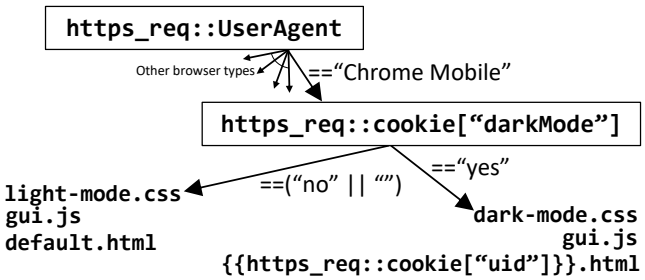


Figure 2: A simplified example of a path constraint tree. At page load time, Oblique's client-side JavaScript library traverses the tree, using load-time concrete values to trace a path to a leaf. The leaf enumerates which URLs Oblique should prefetch. Those URLs may need to be concretized with load-time values (e.g., a cookie value in this example).

In this paper, we introduce Oblique, a new system for accelerating page loads. Oblique's goal is to improve the load time reductions provided by state-of-the-art accelerators, while enabling *cheaper, more secure outsourcing* of the analyses which identify the objects that a client should prefetch. Figure 1 depicts Oblique's architecture. When a content provider creates a new page, the provider feeds the new content to a third-party Oblique server. The server performs a *symbolic page load*, exploring the possible behaviors of a web browser and a web server during the page load process. The output of the symbolic page load is a *path constraint tree*, as shown in Figure 2. Each leaf is a set of URLs that a client should prefetch, and each path from root to leaf represents symbolic constraints on the actual client-side and server-side state that is observed at the time of a real page load. During an actual page load, Oblique inspects concrete state like a client's cookie, traces a path through the constraint tree, and prefetches the relevant objects using a client-side JavaScript library.

Security: Oblique's offline analysis server does not see concrete instances of uniquely-identifying client data. For example, the server does not observe concrete values for any user's cookies or `User-Agent` string. Instead, the analysis server only sees page content that could have been fetched by any actor on the internet who can issue HTTPS requests. Later, during an actual page load, the analysis server is totally uninvolved, and receives no information about sensitive concrete values. In contrast, remote dependency resolution (RDR) forces clients to divulge cleartext HTTPS traffic, exposing cookies and other private values.

Financial cost: Oblique's offline symbolic analysis occurs when a new page version is created. The analysis cost (as measured by VM rental fees) is amortized across all client loads of the page. For popular pages, this amortized expense will be less than the aggregate per-page-load costs incurred by third-party RDR or first-party accelerators like Vroom. RDR must launch a proxy-side web browser for every client-initiated page load, whereas a Vroom-enabled web server must analyze HTML on-the-fly (§5.2).

Performance: In addition to Oblique's security and cost benefits, Oblique also loads pages up to 16% faster than RDR and Vroom (§5.2). The reason is that Oblique's symbolic analysis enables more accurate prefetching: fewer unnecessary objects are prefetched, and more objects that are truly needed are prefetched. For example, Oblique can accurately model URLs that embed random numbers; these numbers are represented symbolically in a path constraint tree, and are late-bound to concrete values at page load time (when a client generates concrete random numbers). In contrast, Vroom and RDR early-bind random numbers at analysis time, resulting in wasted prefetches, or potentially prefetchable objects being ignored (§3.3).

Summary: This paper provides three contributions:

- We describe how to symbolically evaluate client-side page load activity (§3.2), using concolic execution to model the JavaScript engine and the rendering engine. Core technical challenges involve tracking symbols that flow across the DOM interface, and preventing Oblique's third-party server from gaining insights about sensitive client-side values derived from nondeterministic functions like `Math.random()`.
- We also describe how to symbolically evaluate the *server-side* half of a page load. By analyzing both sides of a load, Oblique can generate even better prefetching hints (§3.4). The core challenges involve the complexities of HTML templating engines, and the careful orchestration needed to ensure that server-side symbols propagate to the symbolic analysis of client-side behavior. To analyze both client-side and server-side behavior, Oblique requires access to backend code and data; that state is sensitive, so Oblique should execute on first-party machines in this mode.
- We build and evaluate an Oblique prototype, and compare it to prior state-of-the-art load accelerators. When Oblique executes in third-party analysis mode, it only analyzes client-side symbols; in this mode, first-party developers can securely outsource prefetch analysis to Oblique, enjoying better security than RDR, and faster page loads than both RDR and Vroom. For popular pages, Oblique also provides lower economic costs due to better amortization of analysis overheads. If page owners are willing to run Oblique on first-party infrastructure, Oblique's client+server analysis can unlock even greater reductions in load time.

Oblique requires no changes to end-user browsers, and reduces overall page load times by up to 31%. To the best of our knowledge, Oblique is the first web accelerator that securely enables outsourced prefetch analysis for HTTPS content.

2 Background

A web page's *dependency graph* [21] captures the load-order relationships between a page's constituent objects. For example, a page's top-level HTML might contain references to a JavaScript file and an image. To load the page, a browser must fetch and evaluate both objects. Evaluating the JavaScript file might generate additional fetches, e.g., because the executed JavaScript code uses the `Fetch` API to issue new HTTP requests. Evaluating the image file causes the associated pixels to be displayed; the reception of the image data may also trigger JavaScript `onload` event handlers. Those handlers can generate more fetches. The overall page load completes when a critical subset of a page's objects have been fetched and evaluated. Different load metrics use different criteria to identify the critical subset (§5).

Web accelerators leverage knowledge of a page's dependency graph to reduce a page's load time. One popular approach is remote dependency resolution (RDR) [1, 3, 23, 24, 26, 35]. An RDR system deploys a proxy server that has low-latency paths to the internet core. An end-user's browser sends each page load request to the proxy. Upon receiving such a request, the proxy launches a headless browser (i.e., a browser that lacks a GUI). The proxy-side browser loads the requested page and streams the fetched objects to the user's browser. By doing so, the proxy can partially mask the user's high last-mile latency: the page's dependency graph is resolved via the proxy's fast network links, and the bytes in each discovered object are pushed to the client as soon as the proxy receives those bytes.

RDR can reduce page load times by up to 40% [24]. Unfortunately, RDR proxies are computationally expensive to run, because web browsers (even headless ones) are complex, resource-intensive applications. A proxy can use backwards program slicing [37] to try to only execute the JavaScript code that influences calls to functions like `Fetch()`. However, slices are often inexact, and the degraded prefetching underperforms traditional RDR for 34% of pages [34].

An RDR proxy must act as a man-in-the-middle for TLS connections. Doing so allows the headless browser to parse cleartext web content and fetch the same objects that a user's browser will eventually want to fetch. However, breaking TLS's end-to-end security is obviously problematic; it allows RDR proxies to see user cookies and other sensitive HTTPS content.[1] This security violation also plagues non-RDR accelerators that perform third-party analysis of dependency graphs [6, 22, 40]. Cryptographic schemes that allow middlebox computation over encrypted TLS data [32] are insufficiently expressive to analyze dependency graphs; prefetch analysis requires a Turing-complete language to parse HTML and evaluate JavaScript.

Vroom [29] is a first-party web accelerator: dependency analysis runs on infrastructure belonging to the content provider. For each page, Vroom performs both offline and online analysis. The offline phase runs periodically (e.g., once an hour), using a headless browser to collect the set of URLs loaded by a page. Across multiple offline page loads, Vroom identifies a "stable set" of URLs that were fetched during each load. When a client initiates a real page load, a Vroom-modified web server parses HTML on-the-fly while streaming it to the client, extracting the embedded URLs. These embedded URLs, plus the ones found during offline analysis, comprise the set of URLs to prefetch. The web server induces the client to speculatively load these URLs via a combination of HTTP/2 push [2] and `<link>` prefetch hints [42].

Vroom's analyses run on first-party machines, so HTTPS secrets are not leaked to third parties. However, Vroom's online analysis cannot be outsourced securely: a benevolent mobile provider who wants to run Vroom on behalf of its users will have to break the HTTPS confidentiality of real user page loads. Vroom's offline phase also requires hand-tuning to deal with the heterogeneity of client browsers. For example, many sites define mobile and desktop versions of each page. A server determines which version to return by examining the `User-Agent` header in a client's HTTP request. Vroom's offline phase must be manually configured to explore the state space of all client-specific parameters like `User-Agent` and the client's screen size. Oblique's symbolic analysis allows Oblique to automatically explore this state space.

3 Design

At a high level, Oblique's offline analysis generates a *prefetch tree* for a page. The tree informs a client which HTTPS objects to prefetch in which situations. The input to the tree traversal is client-specific, potentially-sensitive information like cookie values; the output is a set of URLs. Oblique generates the tree by symbolically evaluating the client-side of a page load (§3.2). The URLs (found at the leaf nodes) are symbolic expressions that a client makes concrete by plugging in client-specific information that is never revealed to Oblique. If a page uses Node [25] (a popular server-side JavaScript framework) to generate HTML, Oblique can also symbolically evaluate server-side code (§3.4). Receiving visibility into both client and server execution allows Oblique to generate prefetch trees with more true positives and fewer false negatives: in other words, clients will fetch more useful objects and fewer unnecessary ones.

3.1 Overview of Concolic Execution

Oblique uses a particular variant of symbolic evaluation called concolic execution [14, 31]. In concolic execution, a program is given a concrete set of initial inputs. The program is then executed under the observation of the concolic framework. The concolic framework assigns a "shadow" symbolic expression to each input value and to each internal program variable. An input's initial symbolic expression is only con-

[1] WatchTower [24] allows each HTTPS origin to run its own RDR proxy. This approach solves the security problem by exacerbating the computational overhead problem, since now *every* HTTPS origin must run a proxy.

strained by the limitations of the input's type. For example, a `uint32` input x might receive an initial concrete value of 2, but an initial symbolic constraint of $(0 \leq x \leq 2^{32}-1)$. During the program's execution, the assignment $y = x/2$ would result in y receiving the concrete value of 1, and the symbolic constraint $y == x/2$. When the program's execution hits a branch statement (e.g., `if(x >= 42){...}else{...}`), execution proceeds along the appropriate path, but the symbolic expressions for the branch-test variables are updated. In the running example, the `else` clause is executed because x (equal to 2) is less than 42; x's symbolic constraint is updated to become $(0 \leq x < 42)$. As the program continues execution, variables receive updated concrete values and updated shadow constraints. Eventually, the program halts or a timeout fires. The concolic framework then explores a different execution path by backtracking along the branch history and selecting a branch direction to invert. In the running example, the concolic framework might choose to explore the taken side of the branch `if(x >= 42){...}else{...}`. To do so, the framework inverts the relevant part of x's symbolic expression, generating the constraint $(42 \leq x \leq 2^{32}-1)$. The framework consults an SMT solver [7, 12] to generate a concrete value for x that satisfies the new constraints. Concolic execution then proceeds down the new branch until the program terminates or a timeout fires. This backtrack-and-explore pattern repeats until all execution paths have been discovered or (more likely) the overall time budget for concolic execution expires. For each discovered path, the framework records the *path constraints*, i.e., the symbolic constraints on all of the input variables which must be true for the path to be taken. Note that path constraints are different than the symbolic constraints on a particular variable. In our running example, the constraint on y is $y == x/2$. The path constraints for that execution path are the aggregate set of constraints placed on x and the rest of the program inputs.

3.2 Analyzing Client-side Behavior

In the context of a concolic page load, the program inputs are client-specific environmental variables. These environmental variables determine the content returned by web servers, and the execution paths taken by a page's JavaScript. For example, when a server receives the HTTP request for an HTML file, the server may examine the `User-Agent` header to determine whether to return the mobile-optimized HTML or the desktop-optimized HTML. The value for the local browser's `User-Agent` header is accessible to JavaScript via the `navigator.userAgent` variable; JavaScript code might inspect that variable to execute different code paths for different browsers. Thus, a client's user agent string is an input to the concolic page load. Table 1 enumerates the client-side inputs that Oblique considers.

Figure 3 depicts the life-cycle for a concolic page load. A distributor assigns concrete values to the inputs; the cookie value is set to an empty string, and other inputs are set to

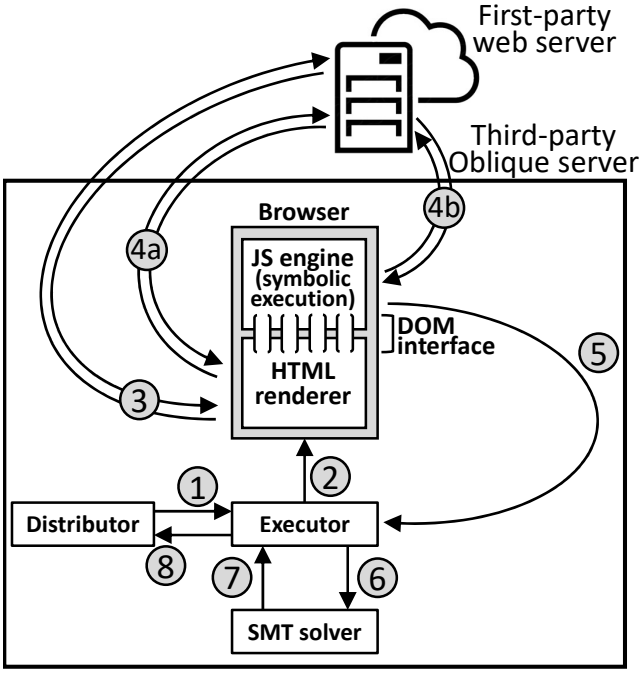


Figure 3: Overview of Oblique's approach for symbolically evaluating a client-side browser. See the mainline paper text for a description of each step.

default values for mobile Chrome. The distributor hands these values to the executor (①). The executor launches a modified web browser (②) that fetches the page's top-level HTML (③). The HTTP request for the top-level HTML uses the environmental values selected by the distributor. Note that the returned HTML will be a concrete string, not a symbolic one.

As the browser parses the HTML, the browser fetches and evaluates non-JavaScript files like CSS and images (④a). When a JavaScript file is fetched (④b), Oblique evaluates it using a modified version of the ExpoSE concolic engine [17]. As the JavaScript code executes, Oblique records the path constraints, and updates JavaScript variables with concrete values and symbolic constraints. When JavaScript code dynamically fetches an HTTP object (e.g., via `fetch(url)`), Oblique uses the *concrete* value of `url` to issue a real fetch. However, Oblique also records the symbolic constraints on `url`. These constraints, which represent a *symbolic URL*, are added to the prefetch list for the current execution path. As a contrived example, a symbolic URL might have the value `"x.com/?{{encodeURI(navigator.userAgent)}}"`; this URL would allow a web server to return different HTML to mobile clients and desktop clients.

In the prior example, the `{{}}` notation indicates a symbolic expression. The example also demonstrates how Oblique is enlightened about certain native functions like `encodeURI()`. Native functions are JavaScript-invocable methods whose implementations are provided by C++ code inside the browser.

Input name	HTTP header	JavaScript variable	Description
User agent	User-Agent	`navigator.userAgent`	The local browser type, e.g., "Mozilla/5.0 (Windows; U; Win98; en-US; rv:0.9.2) Gecko/20010725 Netscape6/6.1"
Platform	Included in User-Agent	`navigator.platform`	The local OS, e.g., "Win64"
Screen characteristics	N/A	`window.screen.*`	Information about the local display, e.g., the dimensions and pixel depth
Host	Host	`location.host`	Specifies the host and port number used by request
Referrer	Referer	`document.referrer`	The URL of the page whose link was followed to generate a request for the current page
Origin	Origin	`location.origin`	Like Referrer, but only includes the origin, omitting path information
Last modified	Last-Modified (response)	`document.lastModified`	Set by the server to indicate the last modification date for the returned resource
Cookie	Cookie (request), Set-Cookie (response)	`document.cookie`	A string containing "key=value" pairs

Table 1: Symbolic inputs to a client-side page load.

Oblique intentionally avoids the concolic execution of native code, since JavaScript-level semantics are the only ones of importance. However, to ensure that native methods correctly propagate JavaScript-level symbolic constraints, Oblique must associate a symbol policy with each native method. A policy describes how the symbolic inputs to a native method should be translated to symbolic outputs for the method. Oblique assigns policies to the most popular native methods that were seen in our test corpus (§5.1). Those methods include the ones defined by the `Math`, `String`, and `RegExp` objects. If a page invokes a native method that lacks a symbol policy, Oblique uses the concrete return value as the symbolic constraint; in other words, the native function acts as a black box that never returns symbolic data.

An HTML renderer maintains an internal data structure called the DOM tree. The DOM tree mirrors the structure of a page's HTML, with each HTML tag having a corresponding DOM node. JavaScript code uses the DOM interface to query or modify the DOM tree, e.g., to implement animations and register event handlers for GUI activity. During a symbolic page load, Oblique associates the DOM tree with a concrete HTML string and a symbolic one; the latter allows JavaScript-level symbols to flow into and out of the DOM tree via DOM methods. For example, given a reference `r` to a `<div>` tag's DOM node, JavaScript code could display the browser type using the assignment `r.innerHTML = navigator.userAgent`. A read of `r`'s parent in the DOM tree (e.g., `r.parentNode.innerHTML`) would return a string whose symbolic value contains `{{UserAgent}}`.

As the page load unfolds, Oblique logs the symbolic URLs that are passed to network APIs like `fetch()`. Oblique also interposes on the DOM interface, and logs the symbolic URLs which cross that interface. For example, suppose that JavaScript code uses the `Node.appendChild(imgNode)` method to add a new `<img>` tag to the page. Oblique would log the symbolic URL associated with the `imgNode.src` attribute; logging the URL reflects the fact that executing `Node.appendChild(imgNode)` causes the browser to fetch an image from a remote server.

Oblique's HTML renderer also logs the static, non-symbolic URLs in a page. These URLs are directly specified in a page's static HTML (e.g., `<link rel="stylesheet" href="styles.css">`) or dynamically injected by JavaScript via the DOM interface. The prefetch list for an execution path contains the static, non-symbolic URLs and the dynamic, possibly-symbolic URLs that are fetched by the path.

Oblique declares the page load to be done when the JavaScript `onload` event fires. The browser fires this event when the browser has finished the HTML parse, fetched all objects discovered by the parse, and evaluated all of those objects. As shown in Figure 3, the JavaScript engine informs the executor about the path constraints for the page load (⑤). The executor asks the SMT solver to invert a branch direction at some point along the path (⑥). Inverting the branch direction changes the symbolic constraints on the input values (§3.1). The SMT solver generates concrete input values that satisfy the new constraints (⑦). The executor returns those concrete

input values to the distributor (⑧). These values represent a new test case that would cause the page to explore a different execution path.

The distributor launches many executors in parallel, running each one on a separate core. As the executors complete and return new test cases, the distributor launches new executors to explore new test cases. The distributor stops creating new executors once a predetermined time budget expires, or there are no more paths to explore. Higher budgets allow Oblique to discover more execution paths, but are more expensive in terms of VM costs. We evaluate these tradeoffs in Section 5.2.

When an executor completes its concolic page load, it logs two things: the list of symbolic URLs fetched by the page load, and the symbolic constraints on client-specific inputs like cookies. Once all executors have finished, the distributor analyses the aggregate set of executor logs to generate a tree of path constraints. Figure 2 provides an example of such a tree. Each leaf contains a set of symbolic URLs; each root-to-leaf path represents the client-specific input values which indicate that a page load will fetch the URLs at the leaf. The distributor translates the constraint tree into a JSON data structure. Finally, the distributor generates a JavaScript library that traverses the tree; at each node, the library applies regular expressions and comparison operators to the JavaScript representation of client-specific inputs (see Table 1). For example, the JavaScript code `/CriOS(54|55)/.test(navigator.userAgent)` determines whether the local browser is Chrome version 54 or 55 that runs atop iOS. Upon arriving at a leaf, the library concretizes the symbolic URLs in the leaf, and then prefetches those URLs using `XMLHttpRequest`.

Oblique sends the prefetching library (which embeds the JSON constraint tree) to the first-party web developer. The developer adds the library as an inline `<script>` tag at the beginning of the associated page's HTML. Later, when a real client browser loads the page, the library issues asynchronous prefetches, populating the local browser cache. As the browser's HTML parse examines the rest of the page and discovers references to external objects, the browser can pull those objects from its cache, avoiding wide-area fetch latencies.

3.3 Nondeterministic JavaScript Functions

JavaScript defines two categories of nondeterministic functions. Timestamp functions like `Date()` and `Performance.now()` read the system clock. Random number generators like `Math.random()` and `crypto.getRandomValues()` create pseudorandom or cryptographically-random byte sequences.

JavaScript code may consult nondeterministic functions during the construction of a dynamic URL. For example, a page might contain code like `if(Math.random() > 0.7){url="a.jpg"}else{url="b.jpg"}`. In that example, the URL embeds no symbols, but its value is controlled by the output of a nondeterministic function. Code like `url=Date() + ".jpg"` would create a URL that directly embeds the output of a nondeterministic function.

Both kinds of dynamic URLs will induce prefetch misses for RDR. The reason is that RDR uses a headless browser to generate a page's dependency graph (§2). The headless browser and the client-side browser will likely generate different nondeterministic values; thus, the two browsers will likely generate different dynamic URLs. To prevent such divergence, RDR could log the nondeterminism observed by the headless browser, and then force clients to use the logged sequence. This approach is the same one used by deterministic replay debuggers to faithfully recreate previously-observed program executions [8, 20]. However, in the context of accelerating page loads, this approach can break functionality. Clients will receive old wall-clock readings, and calculate elapsed time periods that do not accurately reflect the client's true perception of time. As a result, clients may fetch stale content or improperly calculate frame rates for animations. From the security perspective, exposing a client's `crypto.getRandomValues()` sequence to a third party is undesirable, because the client might use the sequence to derive keys or nonces.

Vroom will also suffer prefetch misses for dynamic URLs that are influenced by nondeterministic functions. Vroom's offline analysis identifies a stable set of URLs that are fetched by several different loads of a page (§2). Vroom's stable set analysis will drop URLs that only differ by a timestamp or a random number. The analysis will also drop URLs that do not directly embed nondeterminism, but are fetched via branching paths whose directions are chosen by nondeterminism.

Oblique handles these dynamic URLs without forcing clients to divulge their nondeterminism to third parties. During an offline symbolic execution, Oblique creates a unique, hidden variable for each invocation of a nondeterministic function. Oblique treats this variable as a client-specific input, akin to `document.cookie` or `User-Agent`. This approach enables Oblique to track how the outputs of nondeterministic functions influence branch decisions and the construction of dynamic URLs. For example, suppose that during symbolic execution, a page's JavaScript code invokes `Math.random()` twice, and then calls `Performance.now()`. Oblique generates the hidden variables $\texttt{rand}_0$, $\texttt{rand}_1$, and $\texttt{pnow}_0$. As the symbolic page load continues, the load may generate dynamic URLs like `https://foo.com/?{{rand0}}.js`. Oblique places these URLs in the prefetch list as normal. The symbolic execution may also branch on the values of $\texttt{rand}_1$ and $\texttt{pnow}_0$, just like the symbolic execution might branch on `User-Agent`. Later, during a real client-side page load, Oblique's prefetch library concretizes hidden variables before traversing the path constraint tree. In the previous example, the prefetch library would make two calls to `Math.random()`, and one call to `Performance.now()`. With the hidden variables now concretized, and with client-specific values like `User-Agent` in

hand, the prefetch library can now traverse the path constraint tree and concretize all of the URLs that reside at the appropriate leaf.

The library prefetches the concretized URLs. Finally, the library dynamically patches [20] nondeterministic functions like `Math.random()` and `Performance.now()`, forcing those methods to return the values in the log of concretized hidden variables. The prefetching library is the first JavaScript code that executes in a page. Thus, as the rest of the page's JavaScript code executes, that code will craft dynamic URLs using the same nondeterministic values that Oblique used to construct prefetched URLs.

This approach may still result in unnatural calculations of elapsed time. For example, a page's normal JavaScript code may call `Performance.now()`, execute a lengthy computation, call `Performance.now()` again, and then use the elapsed time to construct a dynamic URL. If Oblique's prefetching library concretizes the two hidden variables using back-to-back calls to `Performance.now()`, the elapsed time used to influence prefetching will be much smaller than the elaspsed time used by the page's normal JavaScript. At worst, this will cause a wasted prefetch; Oblique only prefetches HTTP GET requests which (unlike POST requests) cannot induce side effects on the server. In future work, we hope to devise mechanisms to allow concolic execution to estimate wall clock time. This ability would enable Oblique to concretize hidden timestamp variables with higher fidelity.

JavaScript is an event-driven language. Thus, the execution order of event handlers (e.g., timers and GUI events) is another source of nondeterminism. Oblique does not attempt to control these sources of randomness, because the event loop only goes live after a page's HTML parse completes. This means that event-loop nondeterminism cannot affect URLs fetched during the HTML parse (e.g., via the `.src` attribute of HTML tags, or `XMLHttpRequests` issued by JavaScript). Event-loop nondeterminism *can* affect URLs fetched after the HTML parse completes.

3.4 Analyzing Server-side Behavior

When a web server receives a request for a page's top-level HTML, the server might dynamically construct the returned HTML. For example, the server might inspect the `User-Agent` string in the HTTP request, and return mobile content or desktop content as appropriate. As another example, the server might use the request's cookie to populate the HTML with user-specific URLs, e.g., corresponding to images of a user's previous purchases on an e-commerce site. Oblique's analysis from the previous sections will not detect this potential diversity of embedded URLs. The reason is that the prior analysis assumes that a page has only one version of its top-level HTML, and thus only one set of embedded JavaScript files; if this assumption is true, then the only goal of symbolic analysis is to explore branch paths in the fixed JavaScript code, identifying the dynamically-fetched URLs.

3.4.1 The Workflow

To generate more accurate prefetch lists for dynamically-generated pages, Oblique can optionally perform symbolic execution of both client-side JavaScript (that runs in a browser) and server-side JavaScript (that runs in the Node framework [25]). The end-to-end workflow looks like this:

- **Phase 1:** Oblique first performs a concolic execution of the server-side request handling code. For each test, the inputs are the HTTP request state, as well as nondeterministic function values (e.g., from Node's `crypto.randomBytes()` method). For each concolic path that is explored, Oblique logs the concrete HTML string that is generated, building a *server-side* path constraint tree. Each leaf contains a concrete HTML string, with each root-to-leaf path representing the constraints on server-side inputs that enable the concrete HTML string to be generated.
- **Phase 2:** Each concrete HTML string is fed to the client-side symbolic execution pipeline from Section 3.2. The output of that pipeline is a *client-side* path constraint tree. Each leaf contains symbolic URLs to prefetch, and each root-to-leaf path represents the symbolic constraints on client state that trigger the fetching of the leaf's URLs.
- **Phase 3:** Once Oblique has finished all of the symbolic executions (both client-side and server-side), Oblique creates a "super-constraint tree" which combines the knowledge gleaned from the individual constraint trees. The super tree maps Phase 1 path constraints on server-side inputs to the appropriate client-side path constraint tree from Phase 2; in other words, each leaf in the super tree is a client-side path constraint tree.

When a real client loads the page, the web server uses the values in the HTTP request to traverse the super tree; if the super tree branches on the return values of server-side nondeterministic function, the web server concretizes those values using the approach from Section 3.3. When the server reaches a leaf in the super tree, the server injects the leaf's prefetching library into the dynamically-constructed HTML. The subsequent construction process for the HTML is guided by the values in the HTTP request, and possibly by nondeterministic functions; those functions return the already-concretized values which guided the traversal of the super tree. When the client receives the HTML, Oblique's prefetching library executes as described in Section 3.2.

3.4.2 Templating Engines

In Phase 1, Oblique symbolically executes the server-side request handler. A developer has two options for specifying an entry point into request-handling code. First, a developer can register an `http.Server.request` event handler with Oblique. When a client request arrives, Node creates a new `http.IncomingMessage` object and invokes the handler. Oblique uses the object's HTTP headers as test inputs for concolic execution of the handler.

```
------ Server-side JavaScript ------
app.get('/', function(req, res) {
  //...examine req and derive the template parameters,
  //and then...
  res.render('template.ejs',
             {userAgent: req.headers['user-agent'],
              userID: 'alice',
              userName: 'Alice',
              nonce: random_value});
});
----------- template.ejs -----------
<html>
  <head></head>
  <body>
    <p1> Welcome to foo.com, <%= userName %>! </p1>
    <% if (userAgent.includes('Android')) { %>
      <img src='site-logo-mobile.jpg'>
    <% }  else { %>
      <img src='site-logo-desktop.jpg'>
    <% } %>
    <img id='session-<%= nonce %>'
         src='<%= userID %>.jpg'>
  </body>
</html>
```

Figure 4: An example of dynamic HTML generation using EJS templates. EJS directives are shown in bold.

The disadvantage of the prior approach is that, during the construction of dynamic HTML, a server may consult *IO-based* sources of nondeterminism. For example, the server may issue a database query, or send an RPC to an external server. Oblique does not log and replay such IO responses. Thus, the concretized Phase 1 HTML that Phase 2 consumes may be different than the dynamic HTML that is generated at the time of an actual page fetch. Such a mismatch would hurt Oblique's prefetching accuracy.

Oblique can avoid this problem if server-side code uses a template engine to generate dynamic HTML. For example, consider EJS [9], a popular template framework. EJS defines a `render(html, dict)` method. The first argument is a template string (e.g., "`<html>Hello {{name}} at {{tstamp}}`"). The second argument is a dictionary which maps template arguments to program variables (e.g., `{name: httpReq.cookie.uid, tstamp: Date.now()}`). EJS examines the template and automatically generates a JavaScript program; this program, which is executed by `render()`, performs the necessary computations to parse `dict` and emit the customized HTML. Figure 4 provides a more complex example of an EJS template.

If a developer uses EJS, then she can tell Oblique to concolically analyze the EJS-created templating JavaScript. The output of Phase 1 is now different: it consists of server-side path constraint trees that are associated with just the templating JavaScript, not the overall handler call chain. Each leaf still contains a concrete HTML string that is passed to the concolic client-side analysis in Phase 2. However, a leaf also contains the *symbolic* HTML string that was output by the Phase 1 analysis. The symbols in this string come from the `dict` argument to `render()`. In the example from Figure 4, the symbolic HTML references the `dict` arguments `userName`, `nonce`, and `userID`. Note that the `dict` argument `userAgent` does not appear in symbolic HTML; that argument is branched upon in the path conditions, but is not directly embedded in the HTML itself.

With template integration, Phase 3 is altered as well. When the web server receives a request, the server executes the request handler up to the invocation of `render()`. At that point, the server has queried any sources of nondeterminism (IO-based or otherwise); the server now possesses concrete values for all the inputs to `render()`. The server can then traverse the super tree, find the appropriate symbolic HTML, concretize it, extract the static URLs inside the concrete HTML, and then inject the appropriate prefetching library. Note that extracting static URLs from the concretized HTML is faster than a naïve top-to-bottom HTML parse, since Oblique has a priori knowledge of the offsets where the URLs will be.

3.5 Security Analysis

Oblique's security properties depend on whether symbolic analysis examines only client behavior, or both client and server behavior. Consider the scenario in which Oblique only analyzes client-side activity. In this case, Oblique only requires access to first-party content that is already publicly accessible via first-party web servers. From the perspective of a first-party web server, Oblique's third-party analysis engine looks like a normal end-user browser that issues normal HTTPS fetches. During a concolic page load, Oblique does track symbolic constraints on sensitive user values like cookies and `User-Agent` strings. However, these constraints represent a universe of possible values for sensitive variables; the constraints are insufficiently precise to allow Oblique to determine *the specific sensitive values that belong to a particular user*. For instance, we did empirically find JavaScript code which tested cookies for substrings that were user-agnostic; a common pattern was to inspect a cookie for a string representing the current date. However, JavaScript code did not contain the logical equivalent of a giant regular expression which scanned the local cookie, testing whether the cookie contained any value from an explicit list of valid user ids. Such JavaScript code does not exist because it would allow anyone to download the enclosing JavaScript file and learn all of the valid user ids for a site! Thus, Oblique's symbolic constraints on cookies are insufficient to induce concrete cookie values belonging to specific users. Similarly, if Oblique analyzes a page and determines that a possible load path will target Android users that possess a certain set of screen dimensions, this information does not allow Oblique to infer the screen dimensions and platform value for a particular user.

To analyze server-side behavior, Oblique requires access to server-side code; that code is inaccessible to public web

clients. During the concolic execution of that code, Oblique might also query sensitive databases, or contact sensitive network hosts that are inaccessible to public internet hosts. Thus, if a developer wants Oblique to analyze both client-side and server-side behavior, Oblique should be run on first-party machines. Compared to Vroom (which is also a first-party accelerator), Oblique will provide faster page loads (§5.2).

3.6 Limitations

Oblique is not guaranteed to optimize every object fetch made by every page. For example, during concolic execution, a page's JavaScript may invoke unmodeled native functions, i.e., browser-provided C++ functions for which Oblique lacks a symbolic execution policy (§3.2). If concolic execution reaches one of those functions, Oblique must always treat the return value as fully concrete. Doing so will hurt path coverage if the program later branches on the value, since concrete values cannot be "inverted" to force a new branch direction to be explored.

Even if a page avoids unmodeled native functions, path coverage may suffer when symbolic path constraints are difficult to invert. If the constraint solver times out while trying to generate concrete inputs for a new path to explore, the path will not be explored. If this happens, Oblique can miss opportunities to discover prefetchable URLs. We evaluate Oblique's sensitivity to time-out parameters in Section 5.2.

During concolic execution, Oblique may trigger interactions with external entities. For example, a concolically-executed browser may issue `XMLHttpRequests` to remote servers. Oblique should only be used with pages for which such interactions are idempotent (either literally or for practical purposes). This limitation is shared by all prefetching systems which issue queries to live services to perform content analysis.

4 Implementation

To implement Oblique's symbolic analysis, we modified ExpoSE [17]. ExpoSE performs concolic execution of pure JavaScript code, but does not handle environmental interactions like network IO. We modified ExpoSE to interface with two different environmental interfaces: the Node runtime and the Electron [10] HTML renderer. Oblique uses the Node runtime when analyzing server-side code, and uses the Electron runtime when simulating client-side loads. As explained in Section 3.2, we enlightened ExpoSE to model a DOM tree symbolically, so that JavaScript-level symbolic values can flow into and out of the DOM interface. Our changes to ExpoSE were non-trivial, totalling roughly 4,300 lines of code.

Oblique's client-side prefetching library is small, containing approximately 300 lines of Javascript code. When Oblique runs in third-party mode (§3.2), web servers require no modifications (other than having to include Oblique's prefetching library at the top of each page's HTML). When Oblique runs in first-party mode (§3.4), web servers must be enlightened to traverse the super-constraint tree, concretize nondeterministic values, and interact with Oblique's templating infrastructure. To implement an Oblique-compatible web server, we created a front-end HTTP layer that sat in front of a commodity web server. The front-end layer used the nghttp2 HTTP library and the myhtml HTML parser to implement the activities described above.

5 Evaluation

In this section, we compare Oblique's performance to that of Vroom and RDR, two state-of-the-art accelerators for mobile page loads. Our evaluation primarily focuses on the variant of Oblique that only analyzes client-side behavior, since we can evaluate this variant on a large number of commercial sites. Using a corpus of 200 real pages, we find that Oblique reduces page loads by up to 31%, outperforming Vroom and RDR by up to 17% while also reducing VM costs for popular sites (§5.2). Oblique provides these advantages while also enabling secure outsourcing of prefetch analysis (§5.2). In Section 5.3, we use a site that we control to provide a case study of the benefits of analyzing both client-side behavior and server-side behavior. We demonstrate that, if first parties are willing to run Oblique, they can unlock even greater reductions in load time than what client-only analysis provides.

5.1 Methodology

Our experiments used a Galaxy S10e phone that ran Chromium v78. The browser ran atop Linux on Dex [30], a runtime that enables Samsung phones to execute traditional Linux executables; Linux on Dex made it easier for us to write testing scripts and other experimental infrastructure. We automated the initiation of page loads and the collection of load time metrics using the Browsertime [33] library. Internally, Browsertime manipulated Chrome via Selenium's WebDriver APIs [36, 43].

To test Oblique, Vroom, and RDR with real websites, we built a Mahimahi-style tool [23] to record the objects in live web pages. Afterwards, when our test phone sent an HTTP request to an Oblique web server, a Vroom web server, or an RDR proxy, the web server or proxy responded with recorded content if the request hit in the replay cache; otherwise, the web server or proxy issued a live fetch to the appropriate content server. We ran Oblique and Vroom web servers, and RDR proxies, on a Digital Ocean VM with 8 2.3 GHz cores, 16 GB of RAM, and a 2 Gbps NIC. The RDR proxy used headless Chrome [28] to load pages. Vroom's offline analysis also used headless Chrome. The online Vroom web server was a derivative of nghttp2 [38] that used MyHTML [4] and Katana [27] to parse HTML and CSS.

Our phone had an LTE connection with a round-trip time of 47 ms to our Digital Ocean VM. Our test corpus contained 200 pages from the Majestic Million [19]. We selected the 200 most popular pages for which the RTT between our phone

and a page's web server was less than the the RTT between our phone and our Digital Ocean VM. This setup resulted in conservative estimates of the benefits provided by Oblique, Vroom, and RDR, relative to the baseline scenario in which our phone contacted normal web servers directly. For each combination of <page, load time metric, acceleration technique>, we loaded each page 5 times and recorded the average. By default, Oblique and Vroom pages were loaded one hour after the completion of offline analysis, but we perform sensitivity analysis on this parameter in Section 5.2.

5.2 Client-only Analysis

PLT: We first explored Oblique's performance when only the client side of a page load is analyzed. Figure 5 shows results for the page load time (PLT) metric. PLT, as measured by the time to the browser's `onload` event, captures how long a page needs to fetch and evaluate all objects referenced by a page's static HTML. Note that PLT only waits for some dynamically-generated fetches to complete. In particular, PLT waits for fetches triggered by the insertion of new DOM nodes (e.g., `document.body.appendChild(newImg)`), but not for fetches triggered directly by network APIs like `fetch(url)`. Thus, PLT underestimates the extent to which Oblique, Vroom, and RDR reduce overall fetch latencies for a page.

Figure 5a shows that, for a 47 ms RTT and a cold browser cache, Oblique provided the average page with a 24.1% reduction in PLT, relative to a baseline (i.e., non-accelerated) page load. Oblique reduced PLTs by 17.3% more than RDR, and 5.4% more than Vroom. To explore Oblique's benefits with higher RTTs; we connected the smartphone to a desktop machine via WiFi, and used netem [18] to inject additional latency along the smartphone/desktop link. As expected, Oblique's benefits improved as phone-server RTTs grew, because of the increasing value of hiding last-mile latency. For example, Figure 5c shows PLT results for an emulated RTT of 150 ms. Oblique improved the average baseline PLT by 31.4%, outperforming RDR by 16.3% and Vroom by 6.2%.

A page load's *prefetch hit rate* is the fraction of requested objects that hit in the browser cache due to a successful prefetch. As shown in Figure 6a, Oblique enjoyed better prefetch hit rates than both Vroom and RDR. Indeed, Oblique's primary advantage over Vroom was the ability to successfully prefetch dynamic URLs that embedded nondeterministic symbols (§3.3); this advantage is reflected in Figure 6b.

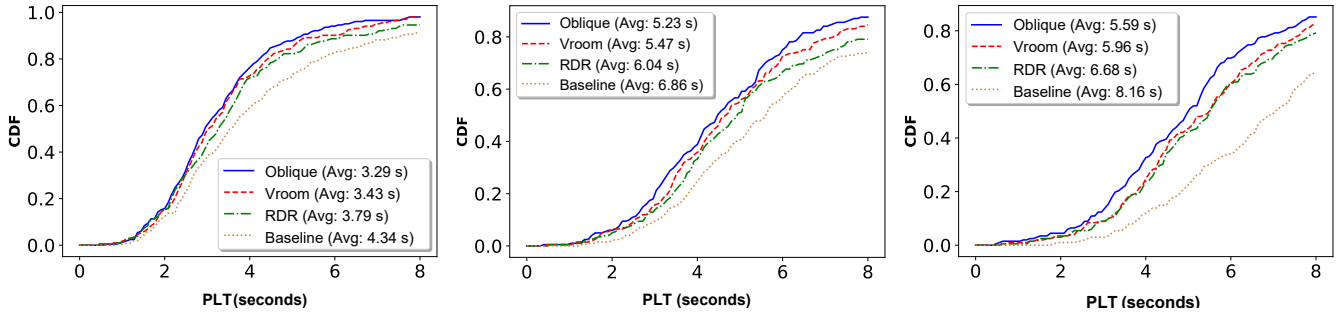
We define a page load's *wasted prefetch rate* as the fraction of prefetched objects that were never requested during the page load. Figure 6c demonstrates that RDR has a much higher percentage of wasted prefetches. The reason is that, for each client-initiated page load, RDR loads the page twice: once on the proxy, and once on the real client machine. Both client-side and server-side nondeterminism may cause the URLs fetched by the proxy's page load to be different than the URLs fetched by the client's browser. Oblique avoids this problem by handling nondeterministic URLs symbolically. In contrast, Vroom's stable-set algorithm simply filters out many nondeterministic URLs. Thus, Vroom has fewer wasted prefetches than RDR, because Vroom does not prefetch nondeterministic URLs that RDR erroneously pulls; however, as shown in Figure 6b, Vroom has a worse hit rate than Oblique due to worse handling of nondeterministic dynamic URLs.

In comparison to RDR, both Oblique and Vroom benefited from informing clients early about the URLs to prefetch. For example, Oblique discovered all of these URLs offline, and prefetched them via the first JavaScript code that executed on a page. Vroom included `<link>` preload tags at the beginning of a page's HTML, and server-pushed other objects to prefetch. In contrast, RDR streamed objects to a client as the proxy discovered those objects; the deeper a page's dependency graph was (§2), the larger the comparative advantage provided by Oblique offline discovery approach. Vroom discovered some prefetch URLs offline, and others during the online, server-side HTML parse. However, Vroom aggressively notified clients about the offline-discovered URLs using server push and `<link>` preload tags.

Warm caches: Figure 7a depicts PLTs for all four systems when browser caches were warm. As expected, all systems enjoyed lower PLTs. Oblique and Vroom had similar performance, but still outperformed RDR.

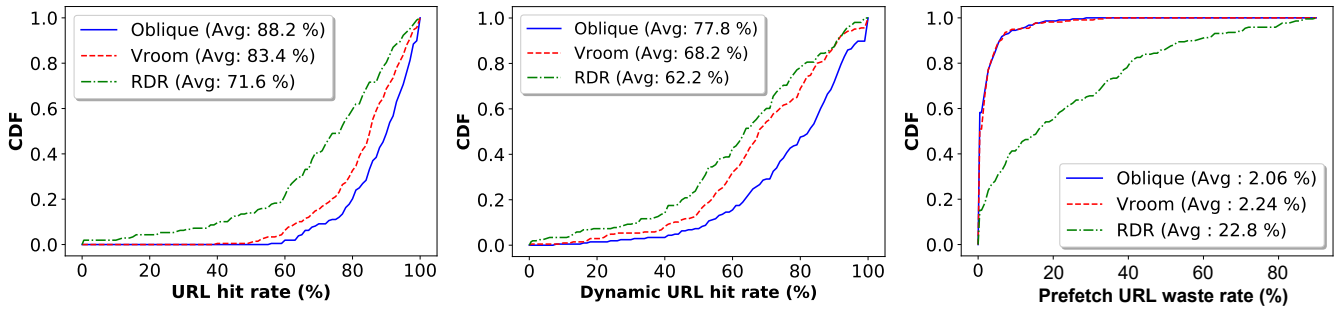
Speed Index: We also evaluated the ability of Oblique, Vroom, and RDR to improve a page's Speed Index [41]. Speed Index is a visual metric that represents how quickly a page's above-the-fold content is rendered. A page's Speed Index is $\int_0^{end} 1 - \frac{p(t)}{100}\,dt$, where *end* is the time of the last pixel change, and $p(t)$ is the percentage of pixels that have already received their final value; lower Speed Indices are better. The formula rewards page loads whose overall rendering time is fast (meaning that *end* values are small). Given two pages with the same *end* value, the formula rewards the page which renders more pixels earlier. Note that Speed Index ignores whether JavaScript files or below-the-fold content has arrived; thus, like PLT, Speed Index underestimates the extent to which accelerators have successfully prepositioned objects.

Figures 7b and 7c show Speed Index results for RTTs of 47 ms and 150 ms. The basic trend is the same one observed for PLT: Oblique has better performance than Vroom, and Vroom has better performance than RDR. However, all of the acceleration systems improve PLT more than Speed Index. For example, with cold caches and a 150 ms RTT, Oblique reduces PLT by 31.4%, but Speed Index by only 20.4%. The reason is that Speed Index only considers visual content, and only cares about the loading of JavaScript files to the extent that the evaluated code modifies a page's above-the-fold graphics. However, deep chains in a page's dependency graph are often caused by JavaScript files whose evaluation triggers the loading of additional JavaScript files [21]. All three accelerators let clients resolve those dependency chains more quickly, but this has less impact on Speed Index than PLT.



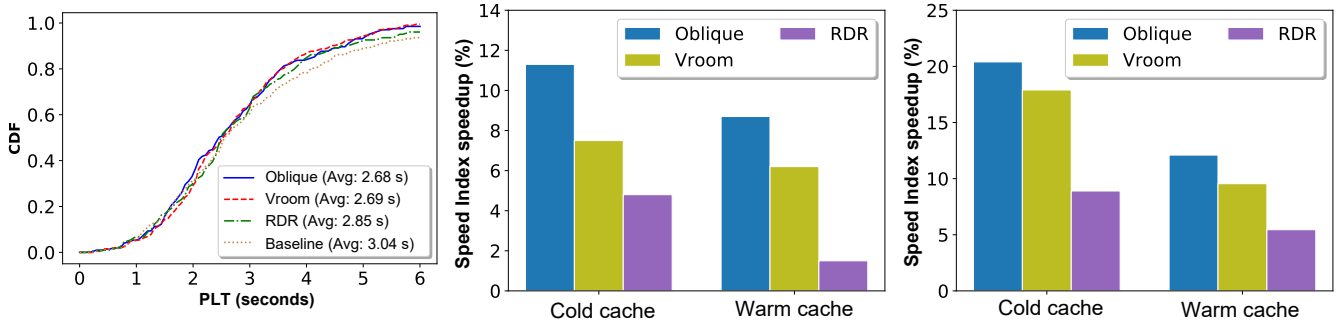
(a) 47 ms RTT **(b)** 100 ms RTT **(c)** 150 ms RTT

Figure 5: Cold-cache PLTs for Oblique, Vroom, RDR, and a baseline, non-accelerated browser.



(a) Prefetch hit rate (static+dynamic URLs) **(b)** Prefetch hit rate (only dynamic URLs) **(c)** Wasted prefetch rate

Figure 6: Prefetch efficiency for Oblique, Vroom, and RDR.



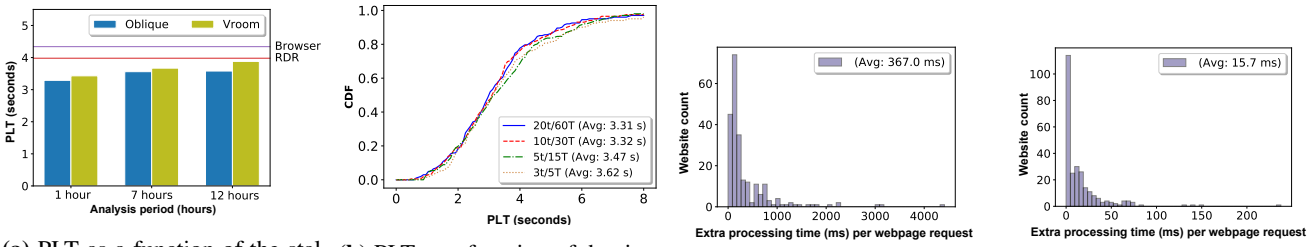
(a) PLT (warm cache, 47 ms RTT) **(b)** Speed Index (47 ms RTT) **(c)** Speed Index (150 ms RTT)

Figure 7: Warm-cache PLTs and Speed Indices (both cold and warm caches). Note that subfigures (b) and (c) have different y-axis scales.

Stale analytic results: In Figures 5 and 7, the offline analyses for Oblique and Vroom occurred one hour before a page load. Figure 8a depicts average PLTs when Oblique and Vroom used analytic data from farther in the pass. Unsurprisingly, Oblique and Vroom performed better with more recent analytic data. However, for up to 12 hours of staleness, Oblique maintained its advantage over Vroom; both Oblique and Vroom also maintained their advantages over RDR and a non-accelerated browser.

Additional analysis time: Given an infinite amount of time, Oblique's offline analysis would be guaranteed to find a complete tree of path constraints; in other words, every possible concretization of client-side symbols would be covered by some root-to-leaf tree path. In practice, Oblique's symbolic analysis is constrained by two parameters: t represents the maximum execution time for a particular execution path,[2] and T represents the overall amount of time that Oblique will analyze the page. By default, Oblique uses $t = 10$ minutes and $T = 30$ minutes. If fully exploring a particular path requires more time than t, Oblique will only discover a subset of the URLs associated with the path. If discovering all paths in a page takes longer than T, Oblique will not generate a prefetch list for the undiscovered paths.

[2] A timeout occurs when the SMT solver cannot negate a branch condition in the current path (§3.1).



(a) PLT as a function of the staleness of the offline analytic data. These results use a 47 ms RTT and cold caches.

(b) PLT as a function of the time budget given to Oblique's offline symbolic analysis. These results use a 47 ms RTT and cold caches.

(c) RDR: Per-page-load computational time required by a proxy.

(d) Vroom: Additional per-page-load computational time required.

Figure 8: The impact of stale Oblique/Vroom analyses, and the additional computational overheads of RDR and Vroom.

Figure 8b depicts Oblique's PLT benefits for different values of t and T. In those experiments, the PLT for a page was defined as the average PLT across all discovered paths; to test the PLT for a particular discovered path, our test browser used concretized client-side symbols that triggered the path. Figure 8b shows that Oblique is basically insensitive to t values above 10 minutes and T values above 30 minutes. The reason is that, for the average page in our test corpus, only 7 minutes were needed to completely explore a path; furthermore, the median page only contained 7 execution paths.

Economic costs: For a given version of a page, Oblique performs an offline analysis once, constructs a path constraint tree, and then incurs no online costs during a real client load. In contrast, RDR must launch an RDR proxy for each client load, and Vroom must perform online HTML parsing. Figures 8c and 8d depict those per-page-load CPU costs.

A VM owner pays for a virtual CPU by the second or by the hour. Once a virtual CPU is fully loaded, any additional computation to perform will force the VM owner to rent more virtual CPU seconds. For a fully-loaded virtual CPU, Vroom requires a VM owner to pay for an additional 15.7 ms of additional compute time per page load. Thus, Oblique's offline analysis becomes cheaper than Vroom's smaller (but repeated) online costs after T/15.7 page loads, where T is Oblique's offline analysis time in units of milliseconds. For example, with a T of 30 minutes, Oblique becomes financially cheaper after 114,650 page loads; with a T of an hour, Oblique becomes cheaper after 229,299 loads. Since RDR imposes much heavier computational overheads than Vroom, Oblique becomes cheaper much faster—after 4,904 loads or 9,809 loads for a T of 30 minutes or a T of an hour. Importantly, these estimates assume that, when a page changes, Oblique's prior analysis is totally invalidated. We are currently investigating how Oblique can use incremental symbolic execution [13, 15] to amortize our analysis costs even more aggressively.

5.3 Oblique in First-party Mode

When Oblique runs on first-party infrastructure, Oblique can symbolically evaluate client-side and server-side behavior. However, to do this, Oblique must be able to examine backend code. We had no access to server-side code for the commercial sites in our test corpus; thus, we had to evaluate first-party Oblique on a collection of modified open-source sites that we ran ourselves. Due to space restrictions, we focus on a single case study of an open-source EJS site. In the text below, Oblique-C refers to a setup in which Oblique can only analyze client-side activity. Oblique-SC refers to a setup in which Oblique can evaluate both server and client behavior.

Gallery Viewer [39] is a site whose core functionality is displaying a rotating set of images. Each image is associated with metadata like an author, a category (e.g., "nature scenes"), and a description of the image; metadata is stored in on-disk JSON files. Users can also chat with each other in real time, and submit comments on particular images. From the perspective of Oblique, the site is interesting because of how it uses cookies and random number generators. The site assigns a unique cookie to each user. When a user requests the page's top-level HTML, the server uses the cookie to query a server-side table of user preferences. The table indicates the types of images that a user likes to view. Given those preferences, the server leverages a random number generator to select random images from the user's preferred image categories. The server inserts the associated image URLs into the dynamically-generated HTML that is returned to the user's browser.

For this particular site, no client-side symbols are relevant to prefetching. However, two kinds of server-side symbols are relevant: the cookie value in the HTTP request for the top-level HTML, and the random numbers that are used to select image URLs.

- Oblique-SC correctly prefetches all of the image URLs. During Phase 1 of analysis (§3.4), Oblique-SC symbolically evaluates the templating JavaScript, creating a symbolic HTML string. In Phase 3, i.e., during a real page load, Oblique-SC runs the server-side event handler up to the call to `render()`. At that point, Oblique-SC concretizes the symbolic HTML using the live cookie data and logged values from the random number genera-

tor. Oblique-SC then extracts the image URLs from the concretized HTML, and creates a prefetch library that downloads those URLs.

- Oblique-C lacks visibility into server-side behavior. Thus, an Oblique-C client incorrectly prefetches the URLs in the concretized HTML that was seen during offline analysis.
- Vroom correctly prefetches the image URLs; the Vroom web server identifies the URLs during the on-the-fly HTML parse.
- RDR incorrectly prefetches the image URLs. The HTML returned to the proxy's headless browser will contain different URLs than the ones in the HTML returned to the user's browser; the URLs in the first HTML file are prefetched by the client.

For a cold browser cache and a 47 ms RTT, Oblique-SC and Vroom had similar performance, with PLTs of 2.01 seconds and 2.06 seconds, respectively. Oblique-C did only slightly better than RDR (2.29 seconds versus 2.37 seconds). The non-accelerated page load required 2.76 seconds.

Oblique-SC has larger computational costs than Oblique-C; during offline analysis, more symbolic execution is required, and during an actual page load, web servers must participate in Phase 3 activity. The extent to which Oblique-SC is preferable to Oblique-C depends on whether first parties want to pay these costs, and the extent to which a site uses server-side symbols to generate HTML. However, the results from Section 5.2 demonstrate that Oblique-C alone can provide impressive reductions in page load time.

6 Conclusion

Oblique is a new system for accelerating mobile page loads. Oblique uses symbolic execution to analyze the various ways that a page load could proceed. For each potential outcome, Oblique creates a list of symbolic URLs that the corresponding page load would fetch. These URLs are concretized at the time of an actual page load, and then prefetched using Oblique's client-side JavaScript library. Oblique works on unmodified browsers, and provides faster page loads than current state-of-the-art approaches. When run in third-party mode, Oblique enables secure outsourcing of prefetch analysis while also enabling reductions in VM costs.

References

[1] Amazon. What Is Amazon Silk?, 2020. https://docs.aws.amazon.com/silk/latest/developerguide/introduction.html.

[2] M. Belshe, BitGo, R. Peon, Google, M. Thomson, and Mozilla. Hypertext Transfer Protocol Version 2 (HTTP/2), May 2015. RFC 7540. https://tools.ietf.org/html/rfc7540.

[3] D. Bhattacherjee, M. Tirmazi, and A. Singla. A Cloud-based Content Gathering Network. In *Proceedings of HotCloud*, Santa Clara, CA, July 2017.

[4] A. Borisov. Fast C/C++ HTML 5 Parser, January 8, 2020. https://github.com/lexborisov/myhtml.

[5] Broadband Search. Mobile vs. Desktop Usage (Latest 2020 Data), 2020. https://www.broadbandsearch.net/blog/mobile-desktop-internet-usage-statistics.

[6] M. Butkiewicz, D. Wang, Z. Wu, H. V. Madhyastha, and V. Sekar. Klotski: Reprioritizing Web Content to Improve User Experience on Mobile Devices. In *Proceedings of NSDI*, Oakland, CA, May 2015.

[7] L. de Moura and N. Bjorner. Z3: An Efficient SMT Solver. In *Proceedings of Tools and Algorithms for the Construction and Analysis of Systems*, Budapest, Hungary, April 2008.

[8] G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai, and P. M. Chen. ReVirt: Enabling Intrusion Analysis through Virtual-Machine Logging and Replay. In *Proceedings of OSDI*, Boston, MA, December 2002.

[9] M. Eernisse. EJS: Embedded JavaScript Templating, 2020. https://ejs.co/.

[10] Electron Community. Electron Documentation, 2020. https://www.electronjs.org/docs/development/v8-development.

[11] F. Rizzato and I. Fogg. How AT&T, Sprint, T-Mobile and Verizon differ in their early 5G approach, February 20, 2020. https://www.opensignal.com/2020/02/20/how-att-sprint-t-mobile-and-verizon-differ-in-their-early-5g-approach.

[12] V. Ganesh and D. L. Dill. A Decision Procedure for Bit-Vectors and Arrays. In *Proceedings of the International Conference in Computer Aided Verification*, Berlin, Germany, July 2007.

[13] P. Godefroid. Compositional Dynamic Test Generation. *ACM SIGPLAN Notices*, 42, January 2007.

[14] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed Automated Random Testing. In *Proceedings of PLDI*, Chicago, IL, June 2005.

[15] P. Godefroid, S. K. Lahiri, and C. Rubio-Gonzalez. Statically Validating Must Summaries for Incremental Compositional Dynamic Test Generation. In *Proceedings of the International Static Analysis Symposium*, Venice, Italy, September 2011.

[16] Google. Google Transparency Report: HTTPS encryption on the web, 2020. https://transparencyreport.google.com/https/overview?hl=en.

[17] B. Loring, D. Mitchell, and J. Kinder. ExpoSE: Practical Symbolic Execution of Standalone JavaScript. In *Proceedings of SPIN*, Santa Barbara, CA, July 2017.

[18] F. Ludovici and H. P. Pfeifer. NetEm - Network Emulator. http://man7.org/linux/man-pages/man8/tc-netem.8.html.

[19] Majestic. The Majestic Million: The million domains we find with the most referring subnets, 2020. https://majestic.com/reports/majestic-million.

[20] J. Mickens, J. Elson, and J. Howell. Mugshot: Deterministic Capture and Replay for JavaScript Applications. In *Proceedings of NSDI*, San Jose, CA, April 2010.

[21] R. Netravali, A. Goyal, J. Mickens, and H. Balakrishnan. Polaris: Faster Page Loads Using Fine-grained Dependency Tracking. In *Proceedings of NSDI*, Santa Clara, CA, March 2016.

[22] R. Netravali and J. Mickens. Prophecy: Accelerating Mobile Page Loads Using Final-State Write Logs. In *Proceedings of NSDI*, Renton, WA, USA, April 2018.

[23] R. Netravali, A. Sivaraman, S. Das, A. Goyal, K. Winstein, J. Mickens, and H. Balakrishnan. Mahimahi: Accurate Record-and-Replay for HTTP. In *Proceedings of ATC*, Santa Clara, CA, July 2015.

[24] R. Netravali, A. Sivaraman, J. Mickens, and H. Balakrishnan. WatchTower: Fast, Secure Mobile Page Loads Using Remote Dependency Resolution. In *Proceedings of Mobisys*, Seoul, South Korea, June 2019.

[25] OpenJS Foundation. Node.js Homepage, 2020. https://nodejs.org/en/.

[26] Opera Norway. Opera Mini, 2020. https://www.opera.com/mobile/mini/android.

[27] QFish. A CSS Parsing Library in Pure C99, 2020. https://github.com/hackers-painters/katana-parser.

[28] J. Ribeiro. Chrome Headless, 2020. Docker Hub. https://hub.docker.com/r/justinribeiro/chrome-headless.

[29] V. Ruamviboonsuk, R. Netravali, M. Uluyol, and H. V. Madhyastha. Vroom: Accelerating the Mobile Web with Server-Aided Dependency Resolution. In *Proceedings of SIGCOMM*, Los Angeles, CA, August 2017.

[30] Samsung. Web Development on a Phone. Updated for Linux on DeX., 2020. https://webview.linuxondex.com/.

[31] K. Sen, D. Marinov, and G. Agha. CUTE: A Concolic Unit Testing Engine for C. In *Proceedings of ESEC/FSE*, Lisbon, Portugal, September 2005.

[32] J. Sherry, C. Lan, R. A. Popa, and S. Ratnasamy. Blind-Box: Deep Packet Inspection over Encrypted Traffic. In *Proceedings of SIGCOMM*, London, United Kingdom, 2015.

[33] Sitespeed.io. Browsertime: Your browser, your page, your scripts, April 15, 2020. https://github.com/sitespeedio/browsertime.

[34] A. Sivakumar, C. Jiang, Y. S. Nam, S. Puzhavakath Narayanan, V. Gopalakrishnan, S. G. Rao, S. Sen, M. Thottethodi, and T. N. Vijaykumar. Nutshell: Scalable whittled proxy execution for low-latency web over cellular networks. In *Proceedings of Mobicom*, Snowbird, Utah, October 2017.

[35] 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 CoNEXT*, Sydney, Australia, December 2014.

[36] Software Freedom Conservancy. SeleniumHQ: Browser Automation, 2020. https://www.selenium.dev/.

[37] F. Tip. A Survey of Program Slicing Techniques. *Journal of Programming Languages*, 3:121–189, 1995.

[38] T. Tsujikawa. Nghttp2 Proxy, 2020. https://nghttp2.org.

[39] R. Villalobos. Building a Website with Node.js and Express.js, 2018. https://github.com/planetoftheweb/expressjs.

[40] X. S. Wang, A. Krishnamurthy, and D. Wetherall. Speeding Up Web Page Loads with Shandian. In *Proceedings of NSDI*, Santa Clara, CA, March 2016.

[41] WebPageTest.org. Documentation: Speed Index, 2020. https://sites.google.com/a/webpagetest.org/docs/using-webpagetest/metrics/speed-index.

[42] World Wide Web Consortium (W3C). Resource Hints, July 2, 2019. W3C Working Draft. https://www.w3.org/TR/resource-hints.

[43] World Wide Web Consortium (W3C). WebDriver, March 27, 2020. W3C Working Draft. https://www.w3.org/TR/webdriver/.