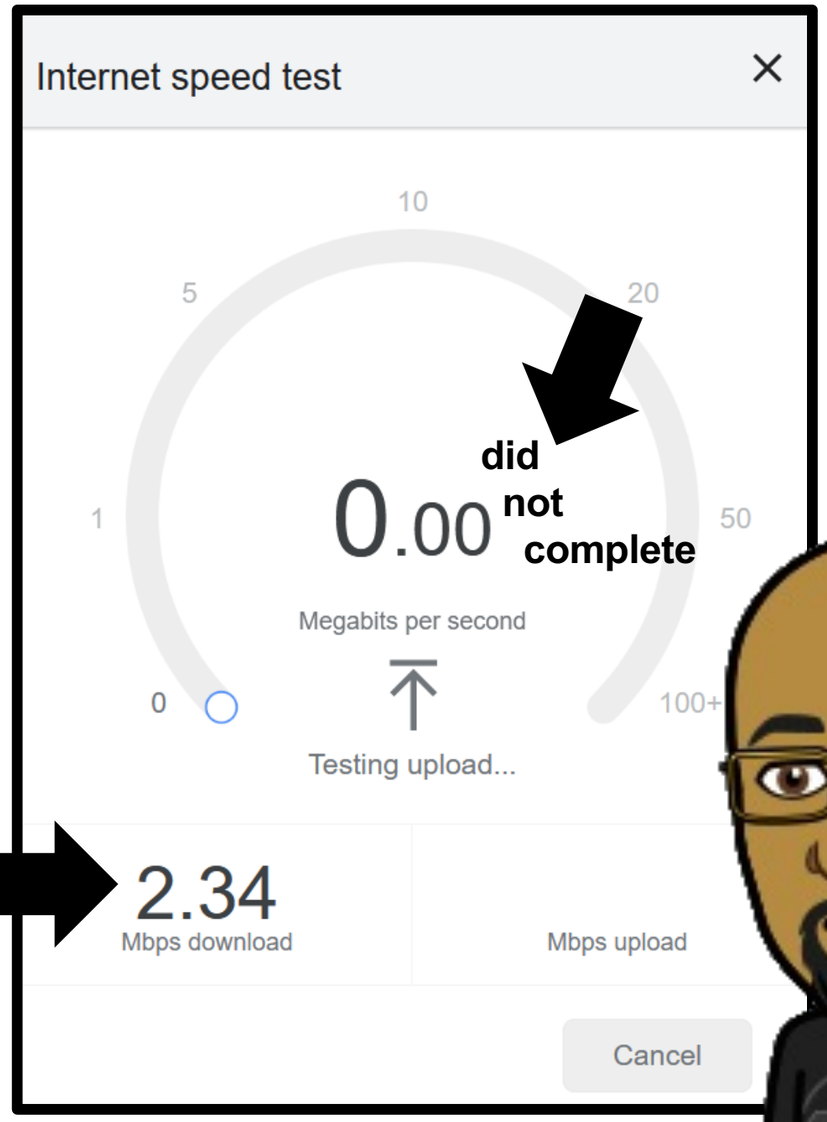
Oblique: Accelerating Page Loads Using Symbolic Execution

nsdi '21

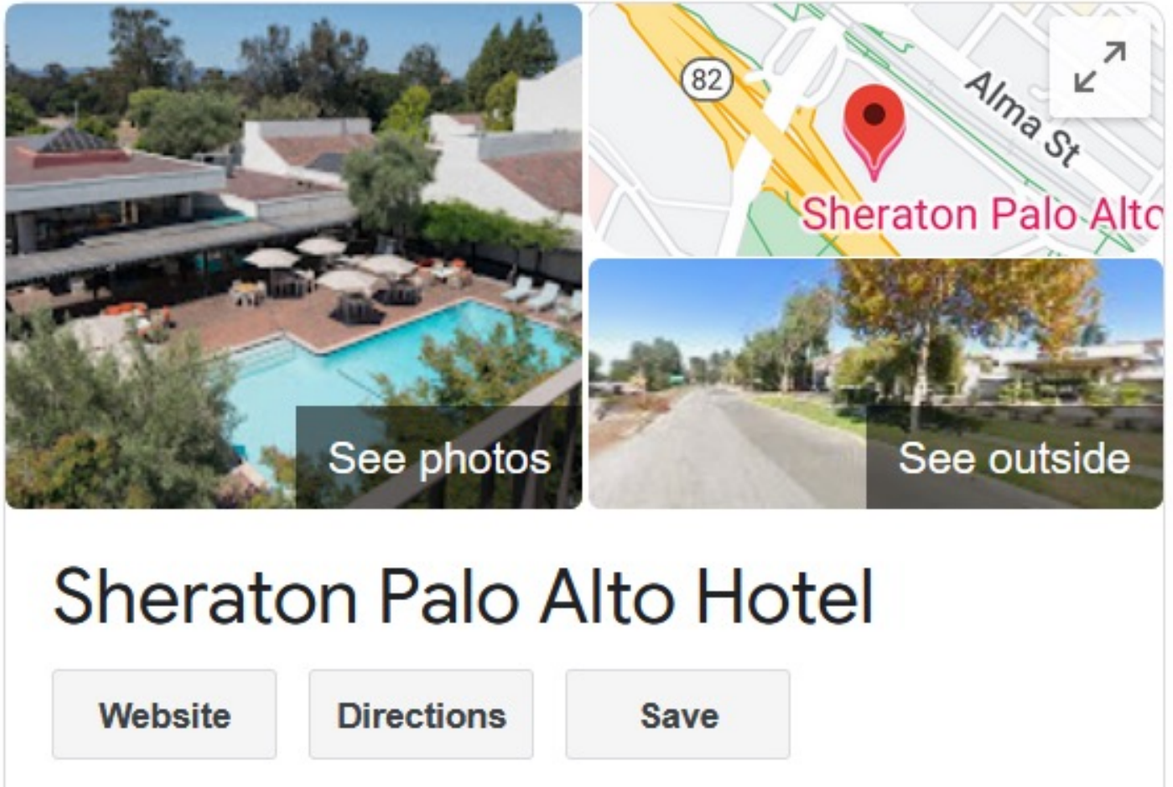
James Mickens
Harvard University

Why I’m Interested in Load Time Optimization



did not complete

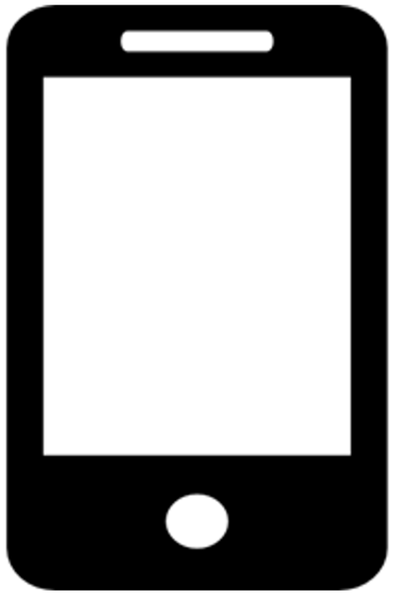
THIS HAPPENED TO ME LAST NIGHT



Outline

- How A Browser Loads a Web Page
- Impediments to Optimization
- Oblique
- What I Learned While Working On Oblique

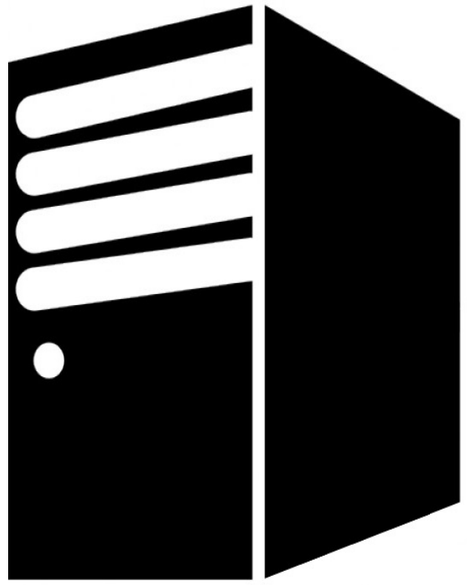
How Does A Browser Load a Page?



User phone

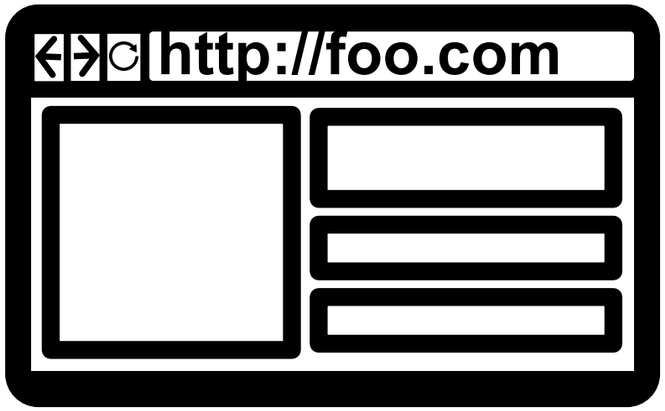
GET

```
<html>
  <head>
    <title>Hi!</title>
    <link rel="stylesheet"
          href="style.css">
    <script src="foo.js"/>
  </head>
<body>
  <div>
    <p>Some content</p>
    <img src="cats.jpg"/>
    <img src="dogs.jpg"/>
  </div>
</body>
</html>
```

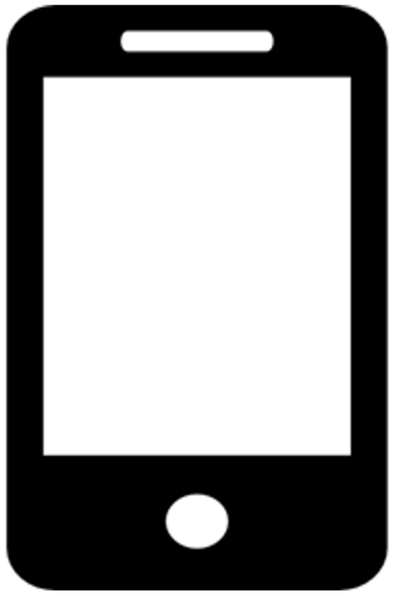


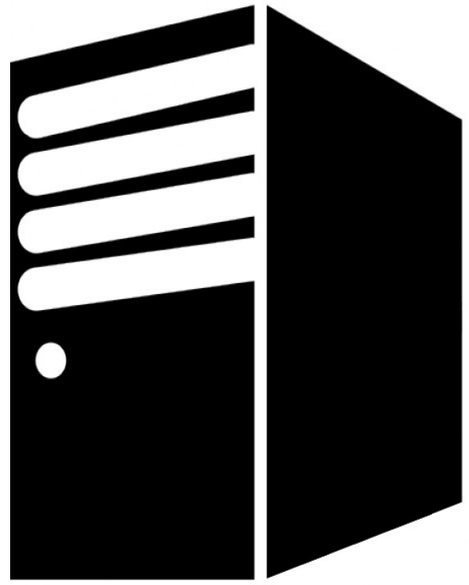
Web server

How Does A Browser Load a Page?



- The HTML contains tags that reference external objects
- To fully load the page, the browser must fetch all of these external objects!

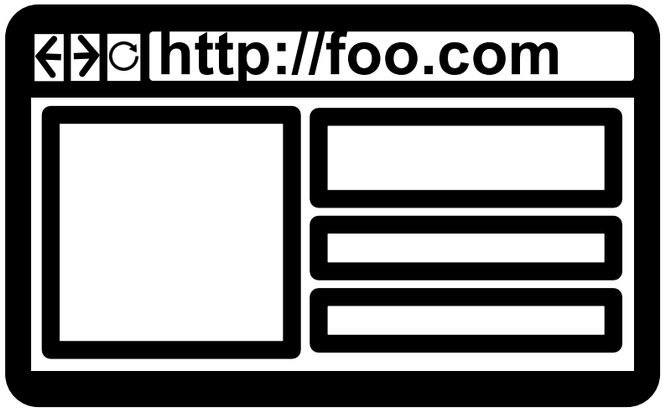




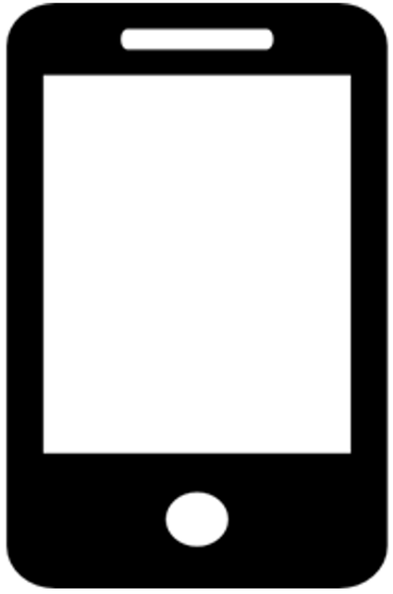
User phone

Web server

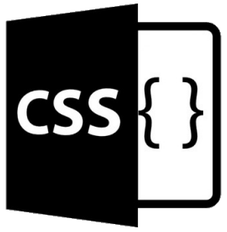
How Does A Browser Load a Page?

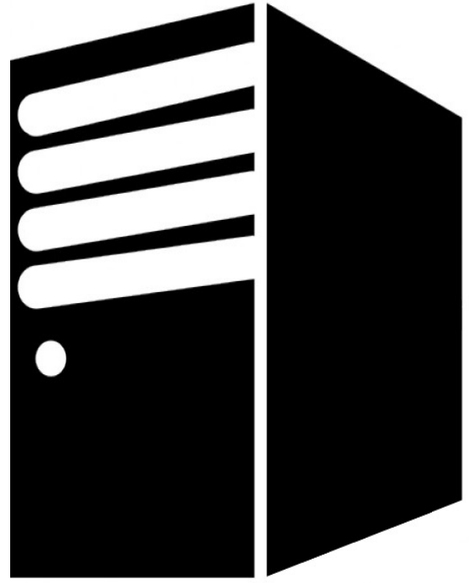


- The HTML contains tags that reference external objects
- To fully load the page, the browser must fetch all of these external objects!



GET http://foo.com/c.css

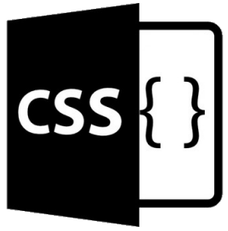


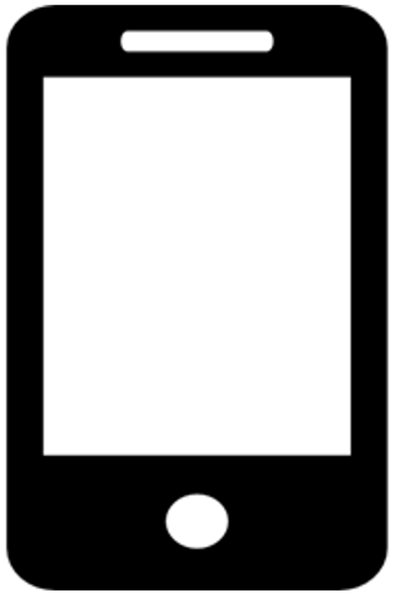


User phone

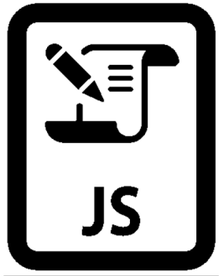
Web server

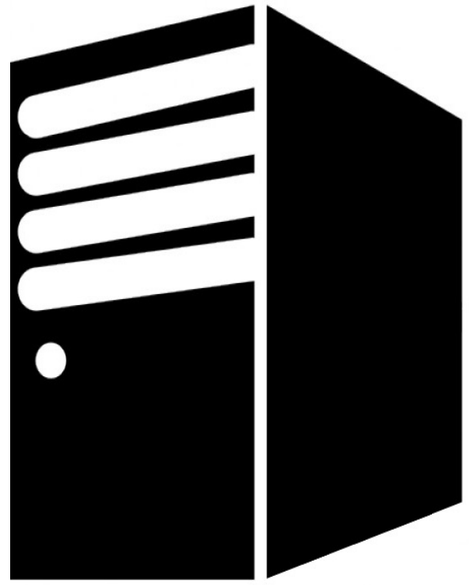
How Does A Browser Load a Page?





GET http://foo.com/first.js

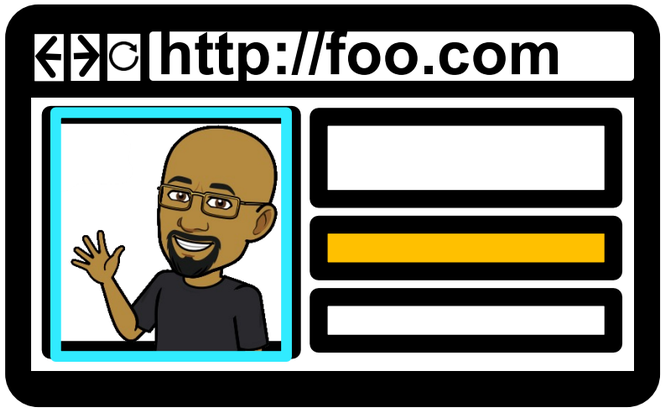


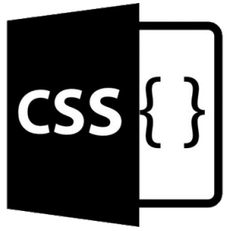


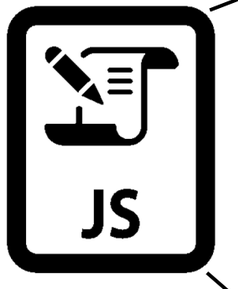
User phone

Web server

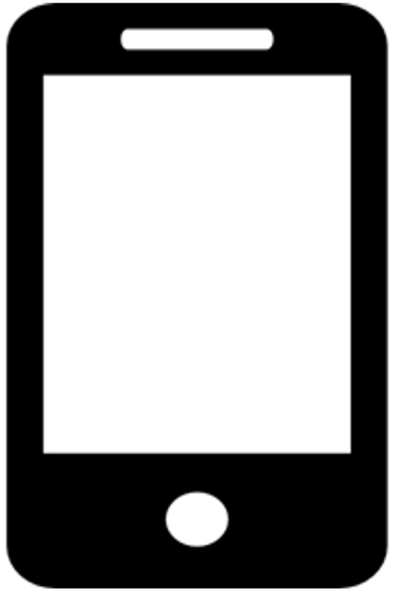
How Does A Browser Load a Page?





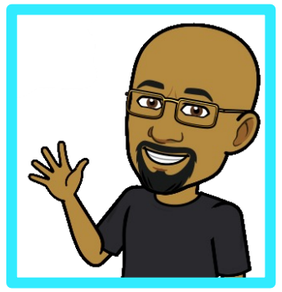


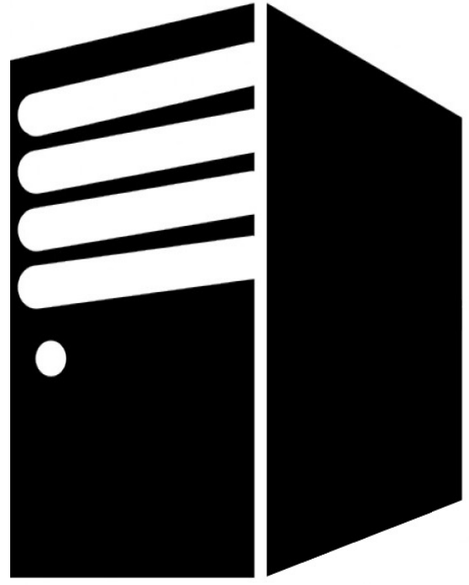

```
if(cookie.loggedIn){
    fetch("http://foo.com/" +
            cookie.userId + ".jpg");
}else{
    fetch("http://foo.com/defaultAvatar.jpg");
}
```



[IF-BODY TAKEN]

GET http://foo.com/42.jpg

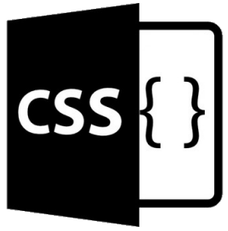


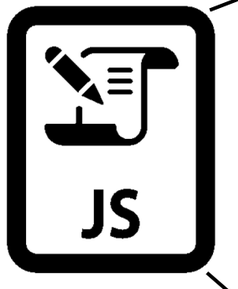


User phone

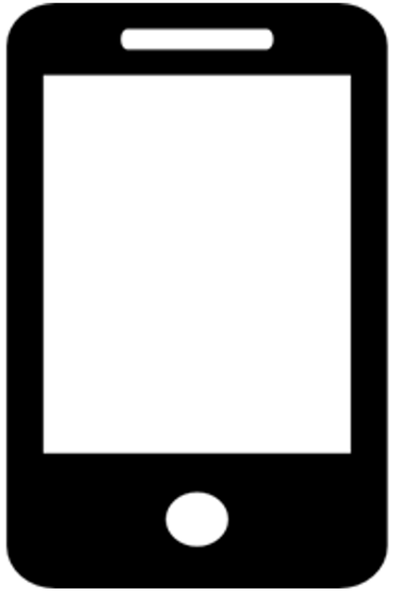
Web server

How Does A Browser Load a Page?



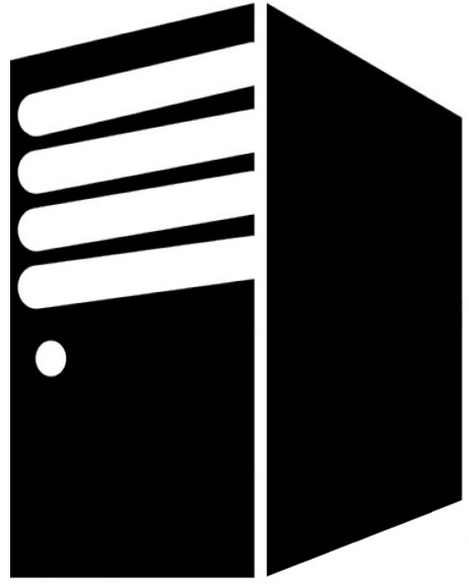



```
if(cookie.loggedIn){
    fetch("http://foo.com/" +
            cookie.userId + ".jpg");
}else{
    fetch("http://foo.com/defaultAvatar.jpg");
}
```



[ELSE-BODY TAKEN]

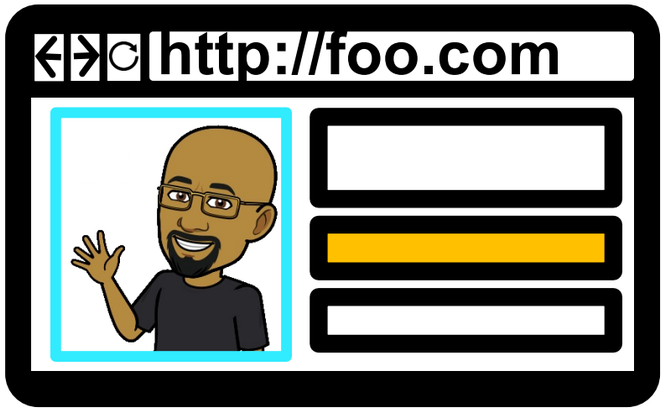
GET http://foo.com/defaultAvatar.jpg

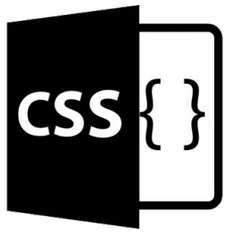


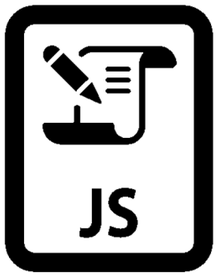
User phone

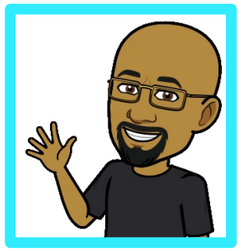
Web server

How Does A Browser Load a Page?



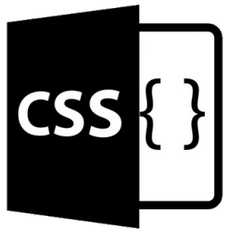


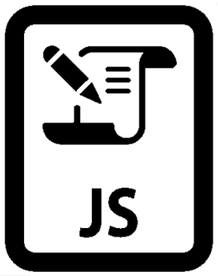




Universe 1

- Some objects to fetch are **statically known**, given the HTML
- The if/else JS statement created two different **universes** because of **dynamic content resolution**
- Branches may check personalized state like a cookie!
- Some dynamically-selected content may be personalized!



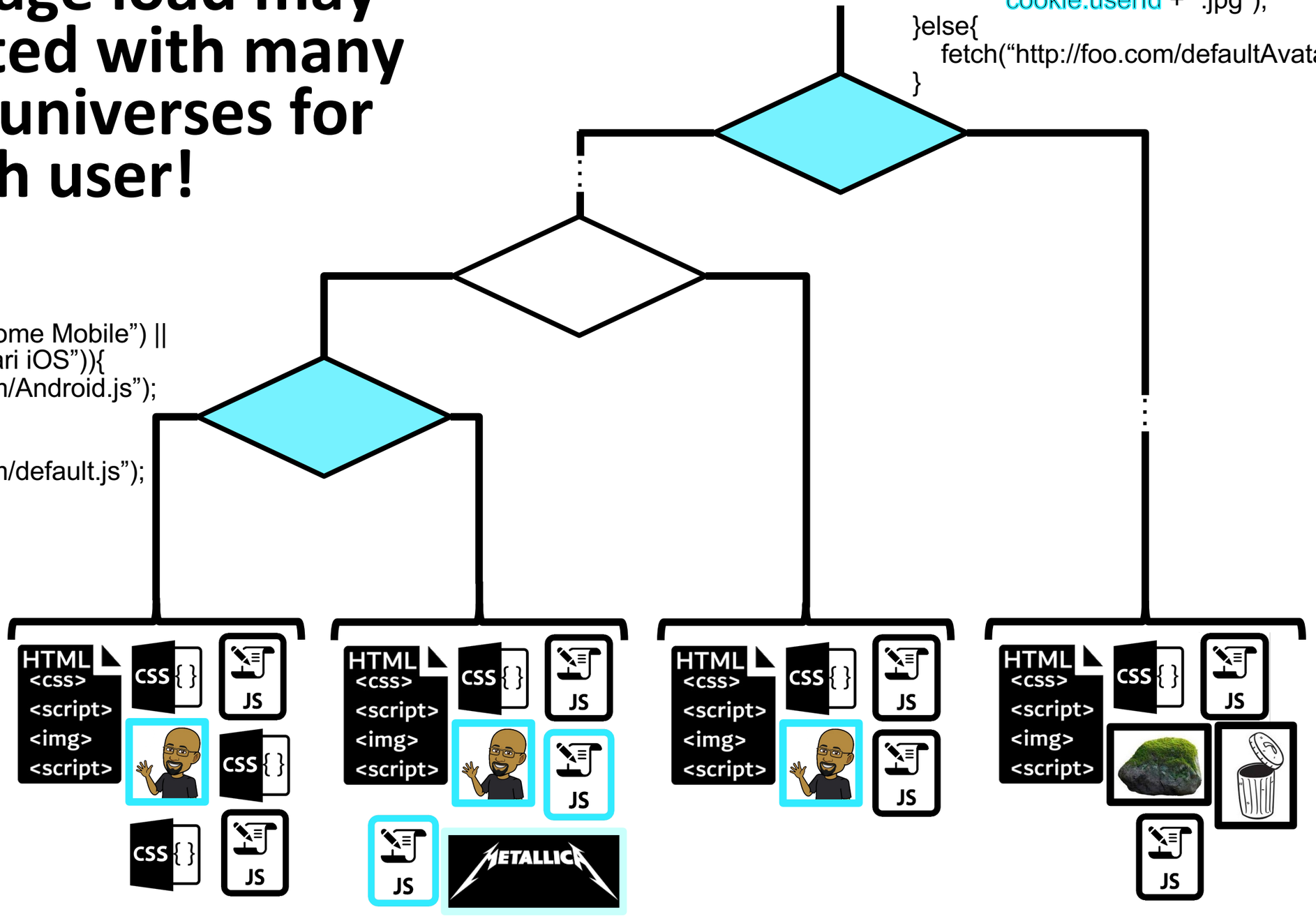


Universe 2

A single page load may be associated with many potential universes for each user!

```
if(cookie.loggedIn){
    fetch(“http://foo.com/” +
          cookie.userId + “.jpg”);
}else{
    fetch(“http://foo.com/defaultAvatar.jpg”);
}
```

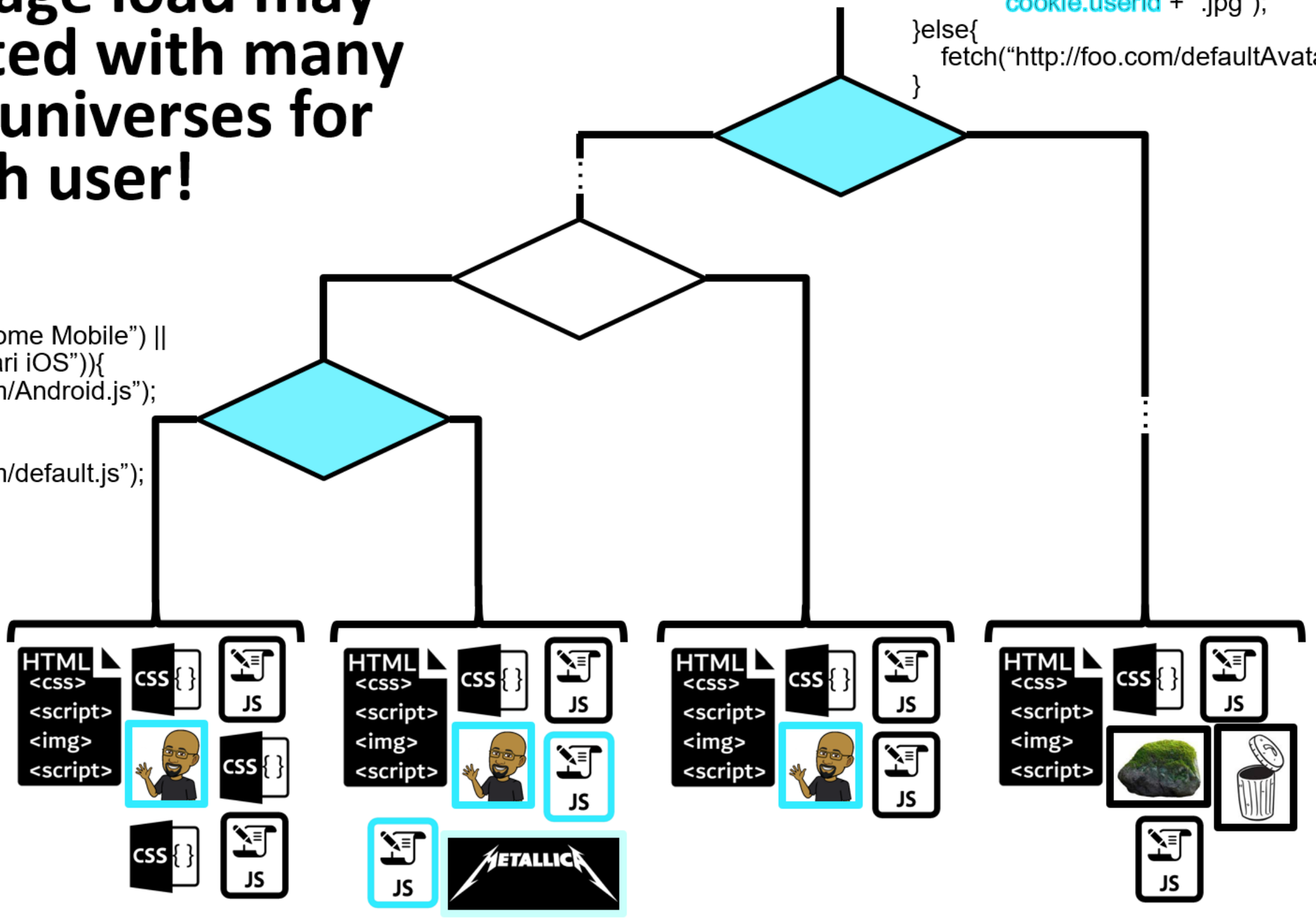
```
if((userAgent == “Chrome Mobile”) ||
   (userAgent == “Safari iOS”)){
    fetch(“http://foo.com/Android.js”);
    fetch(…);
}else{
    fetch(“http://foo.com/default.js”);
    fetch(…);
}
```



A single page load may be associated with many potential universes for each user!

```
if((userAgent == “Chrome Mobile”) ||
   (userAgent == “Safari iOS”)){
    fetch(“http://foo.com/Android.js”);
    fetch(…);
}else{
    fetch(“http://foo.com/default.js”);
    fetch(…);
}
```

```
if(cookie.loggedIn){
    fetch(“http://foo.com/” +
          cookie.userId + “.jpg”);
}else{
    fetch(“http://foo.com/defaultAvatar.jpg”);
}
```

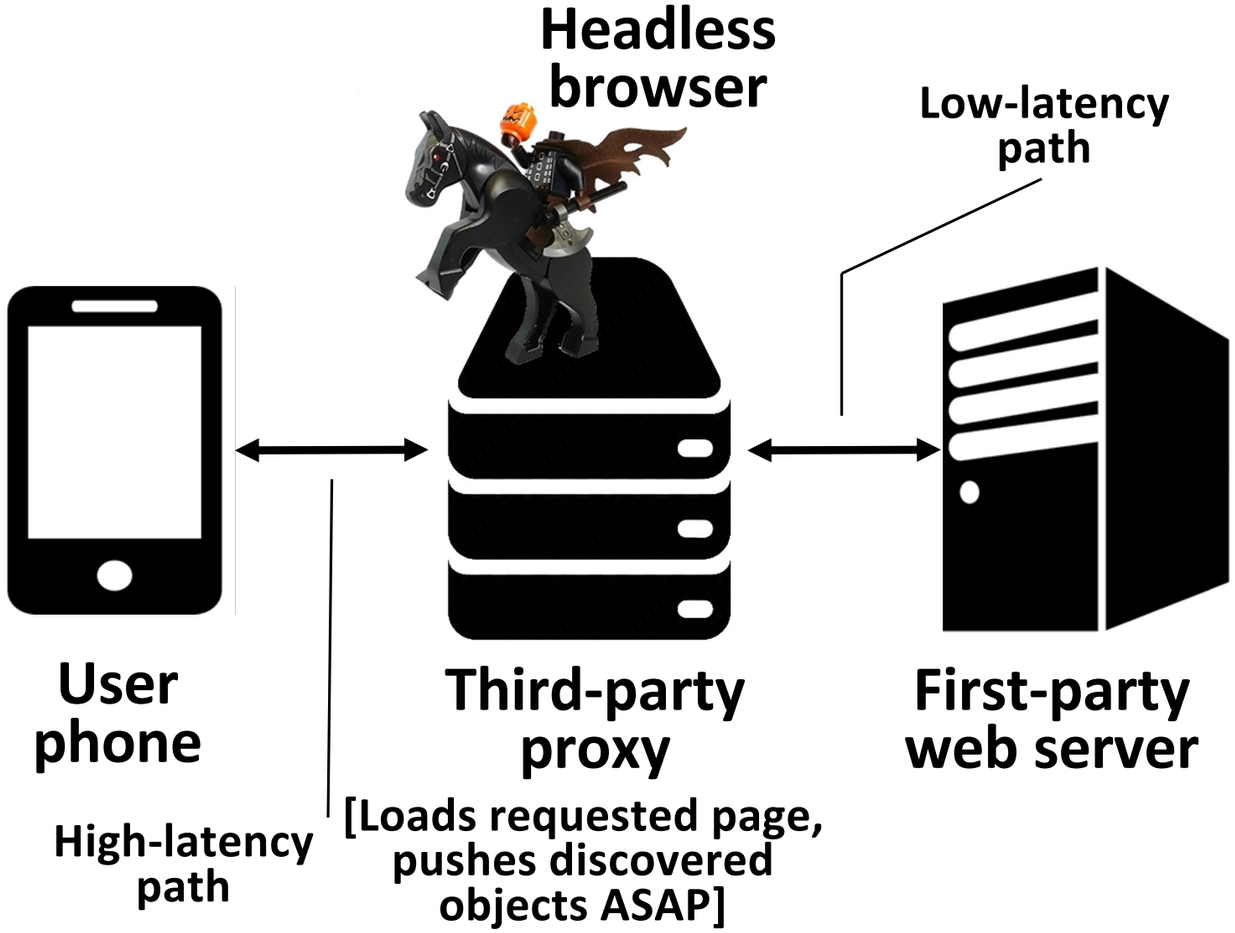


Outline

- How A Browser Loads a Web Page
- Impediments to Optimization
- Oblique
- What I Learned While Working On Oblique

Two Problematic Trends in Web Traffic

- Mobile traffic is growing (> 50%)
  - Many mobile users (particularly in emerging markets) stuck behind high-latency 3G/4G links
  - Even 5G links often suffer from 4G latencies
  - Latency, not bandwidth, often determines page load times!
- Traffic is shifting to HTTPS (> 90%)
  - The crypto is cheap . . .
  - . . . but how can we accelerate encrypted mobile traffic while preserving confidentiality and integrity?

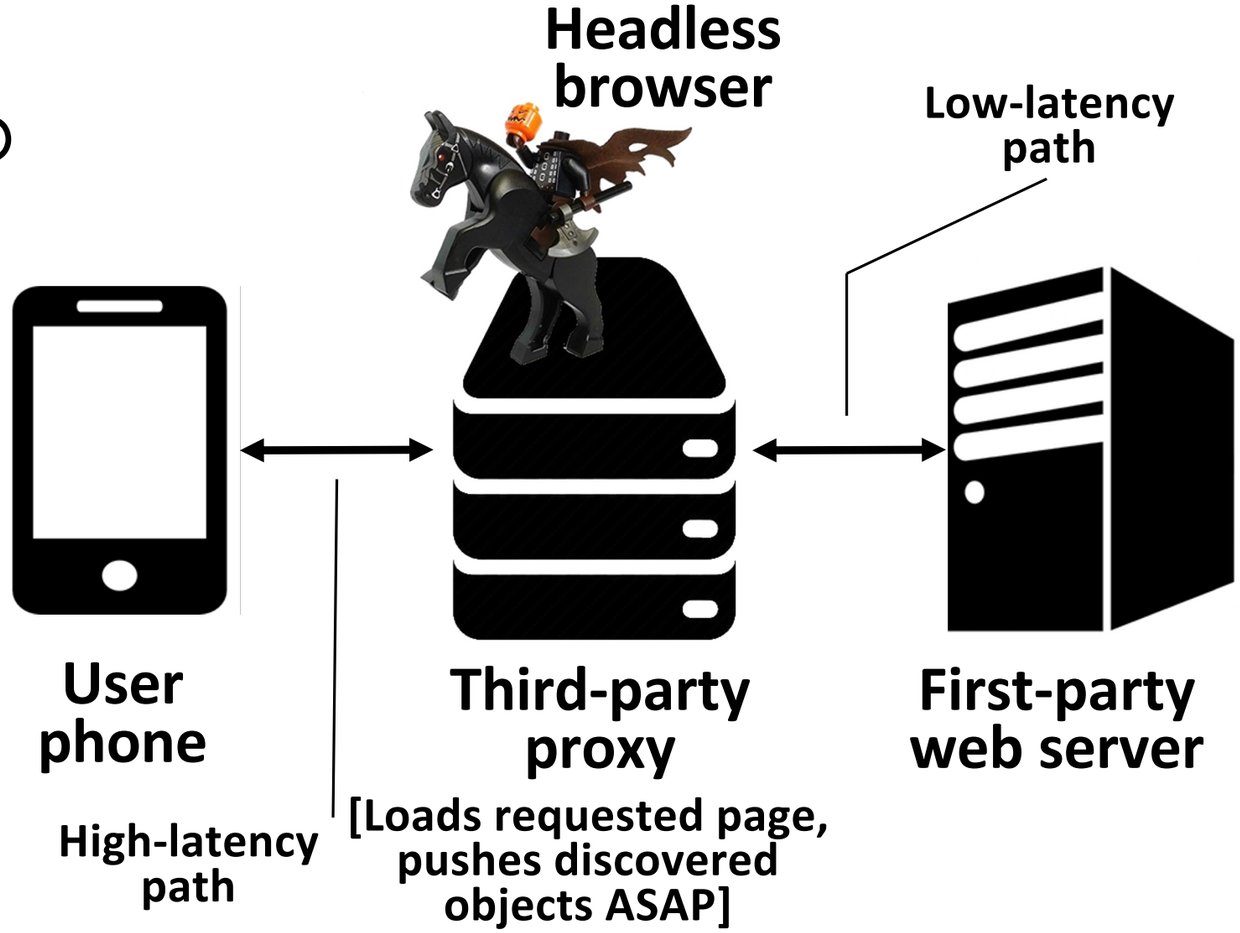


Remote Dependency Resolution (e.g., Amazon Silk, Parcel)

Two Problematic Trends in Web Traffic

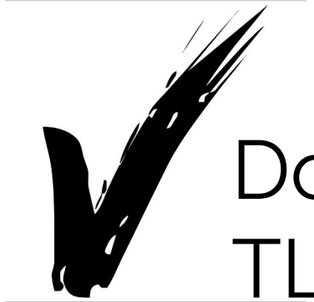
✓ Enables outsourcing of web acceleration

✗ Breaks end-to-end TLS security: cleartext user data (e.g., cookies and **User-Agent** string) are exposed to third party

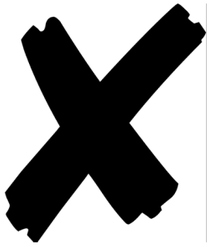


Remote Dependency Resolution (e.g., Amazon Silk, Parcel)

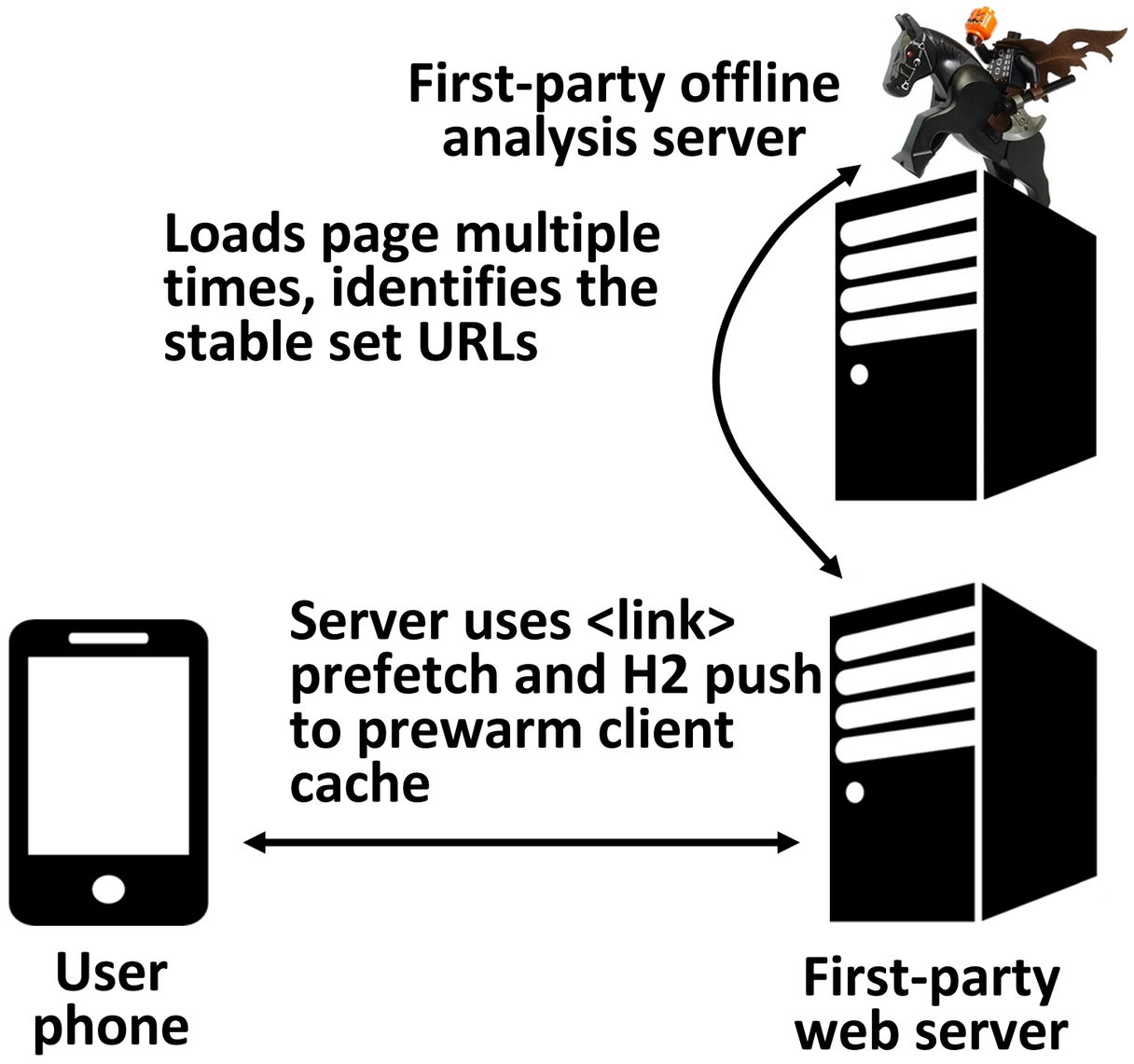
Two Problematic Trends in Web Traffic



Doesn't expose cleartext TLS data to third-party origins



Analysis must be run by the first party: outsourcing would break TLS security



Vroom (SIGCOMM 2017)

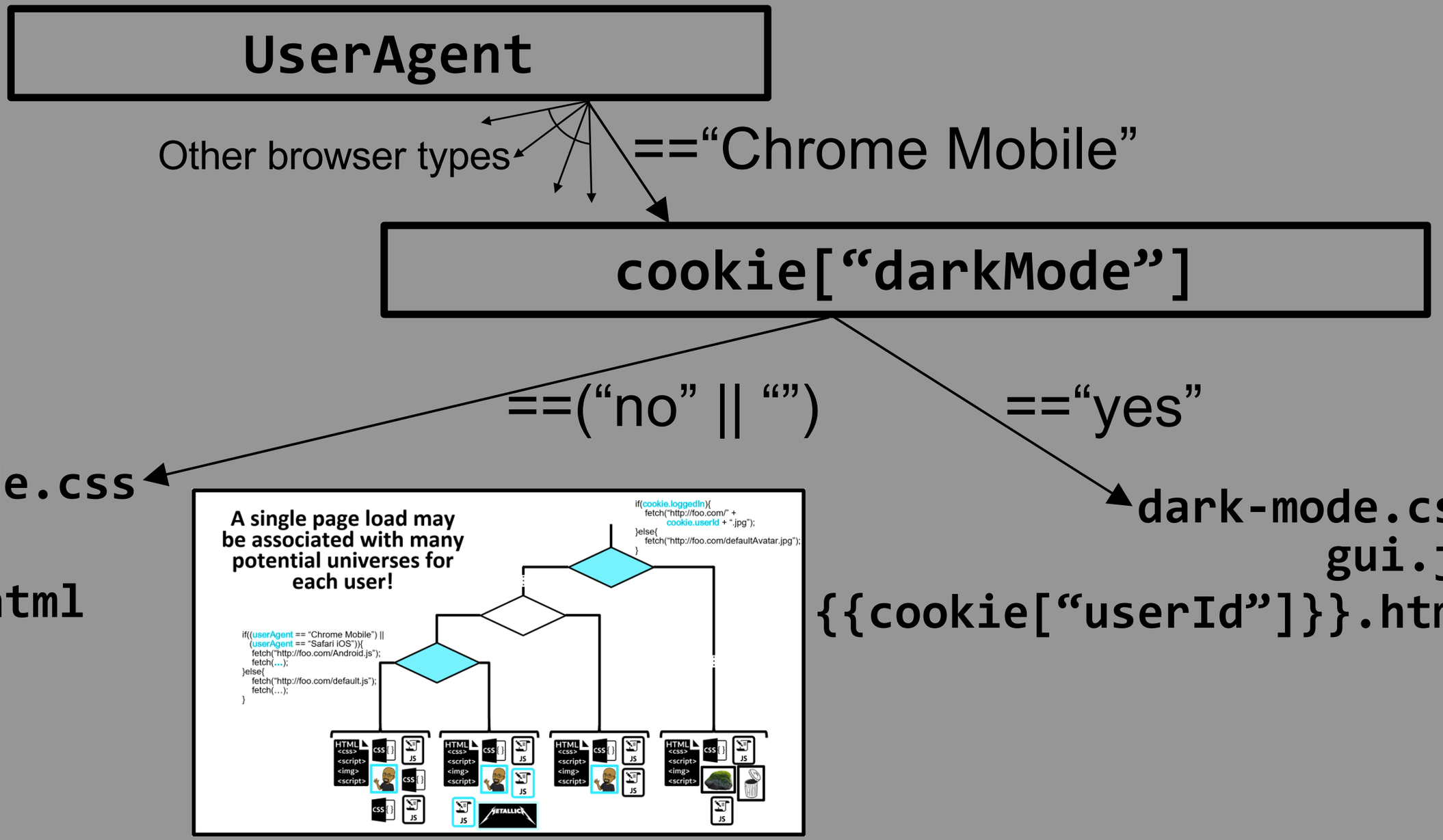
Outline

- How A Browser Loads a Web Page
- Impediments to Optimization
- Oblique
- What I Learned While Working On Oblique

Oblique: The Big Idea™

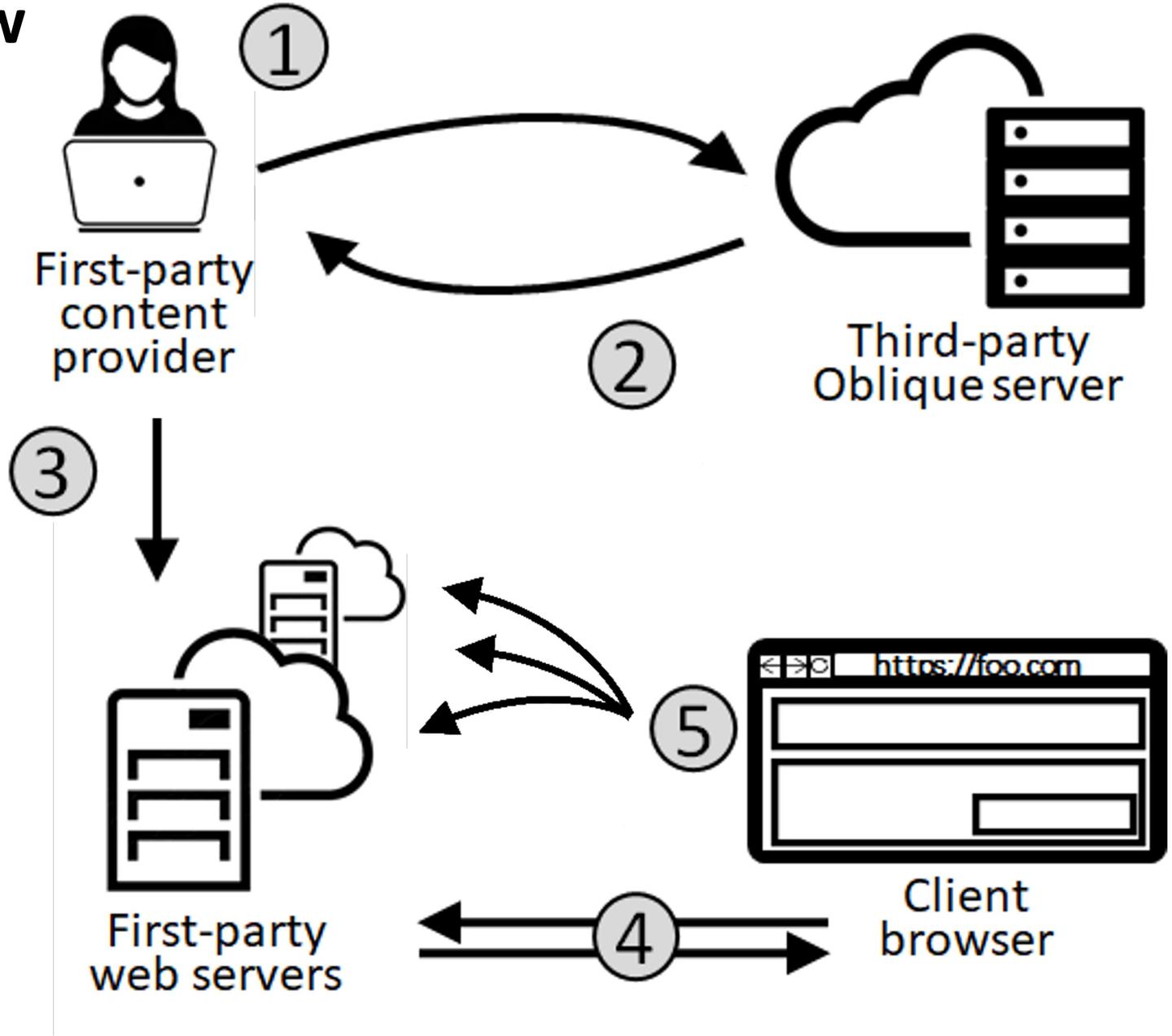
- An offline third-party server loads a web page **symbolically**
  - The symbols are sensitive user values like cookies and **User-Agent** strings
  - Output of analysis is a list of symbolic URLs (e.g., **{{cookie[“userId”]}}.html**) fetched by each universe
- Have the user’s browser:
  - Concretize symbolic client state (thereby picking a universe)
  - Concretize the symbolic URLs
  - Prefetch the symbolic URLs
- User-specific data is never revealed to the third party!

Output of Symbolic Page Load: A Path Constraint Tree



Oblique: End-to-end Workflow

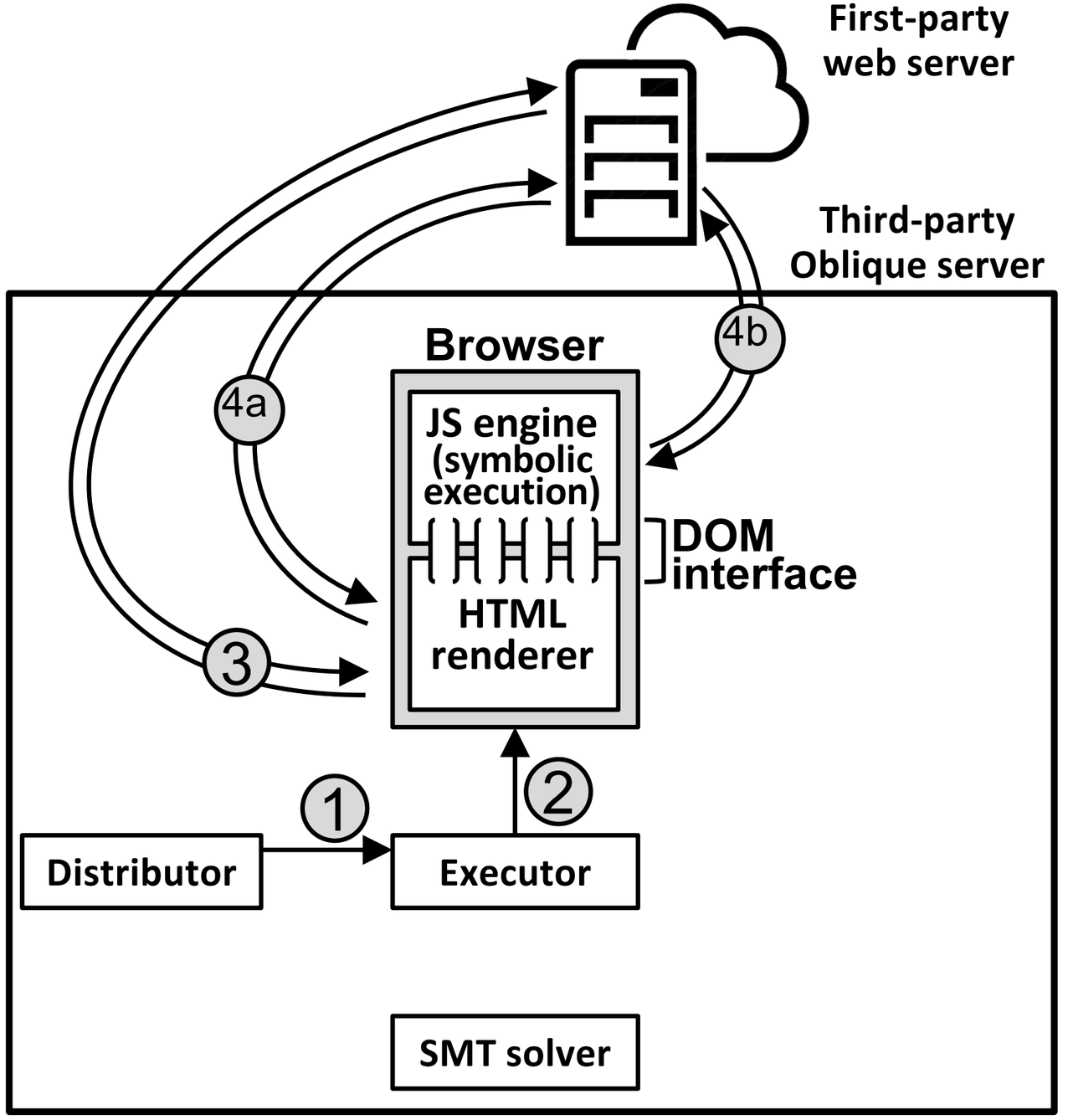
1. Developer uploads page content to Oblique's third-party analysis server
2. Oblique returns a path constraint tree for the page
3. Developer uploads page content+path constraint tree to first-party web servers
4. Later, user fetches the page's HTML+path constraint tree
5. Oblique's JavaScript library concretizes path constraint tree, prefetches objects



teach me
this symbolic analysis
of which thee speak

Symbolic Analysis

1. Distributor generates initial concrete values for client symbols (e.g., Cookie="cat=yes", User-Agent="MobileChrome")
2. Executor launches a web browser
3. Browser fetches concrete page HTML from first-party servers
4. Browser fetches more concrete objects
    a. CSS and images handled as normal
    b. JS evaluated using a concolic engine



Symbolic Analysis

1. Distributor generates initial concrete values for client symbols (e.g., Cookie="cat=yes", User-Agent="MobileChrome").
2. Executor launches a web browser
3. Browser fetches concrete page HTML from first-party servers
4. Browser fetches more concrete objects
    a. CSS and images handled as normal
    b. JS evaluated using a concolic engine

As JS executes on concrete data, Oblique tracks symbolic path constraints and symbolic URLs!

```
var baseUrl = "foo.com/";
var rndId = Math.random().toString();
if(document.cookie.indexOf("cat")==0){
   fetch(baseUrl + rndId + "/cat.jpg");
}else{
   fetch(baseUrl + "/dog.jpg");
}
```

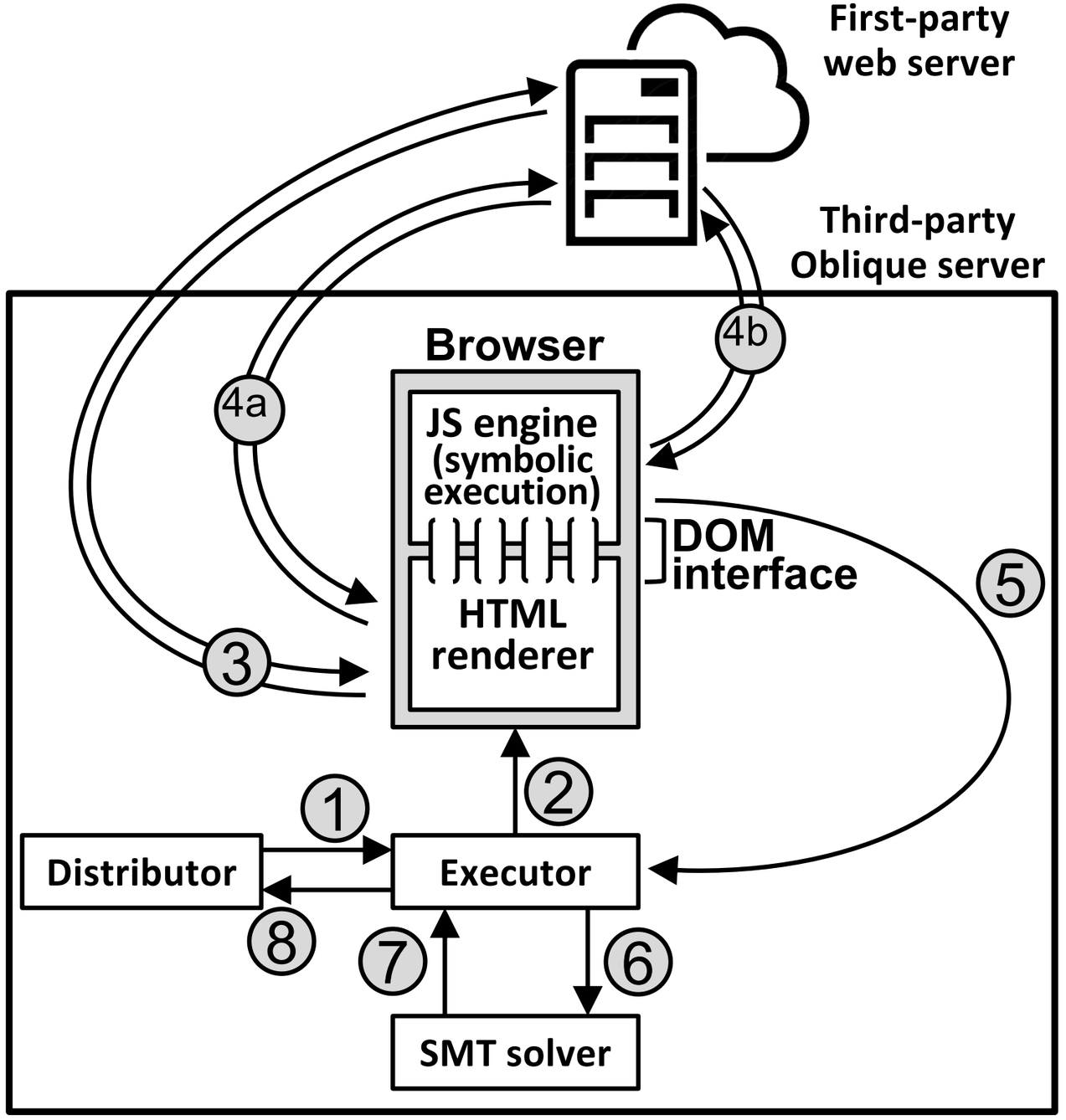
Concrete URL: foo.com/0.3274/cat.jpg

Symbolic URL:
foo.com/{{rnd_0}}/cat.jpg
Symbolic path constraint:
document.cookie = "cat"{{\w*}}

Symbolic Analysis

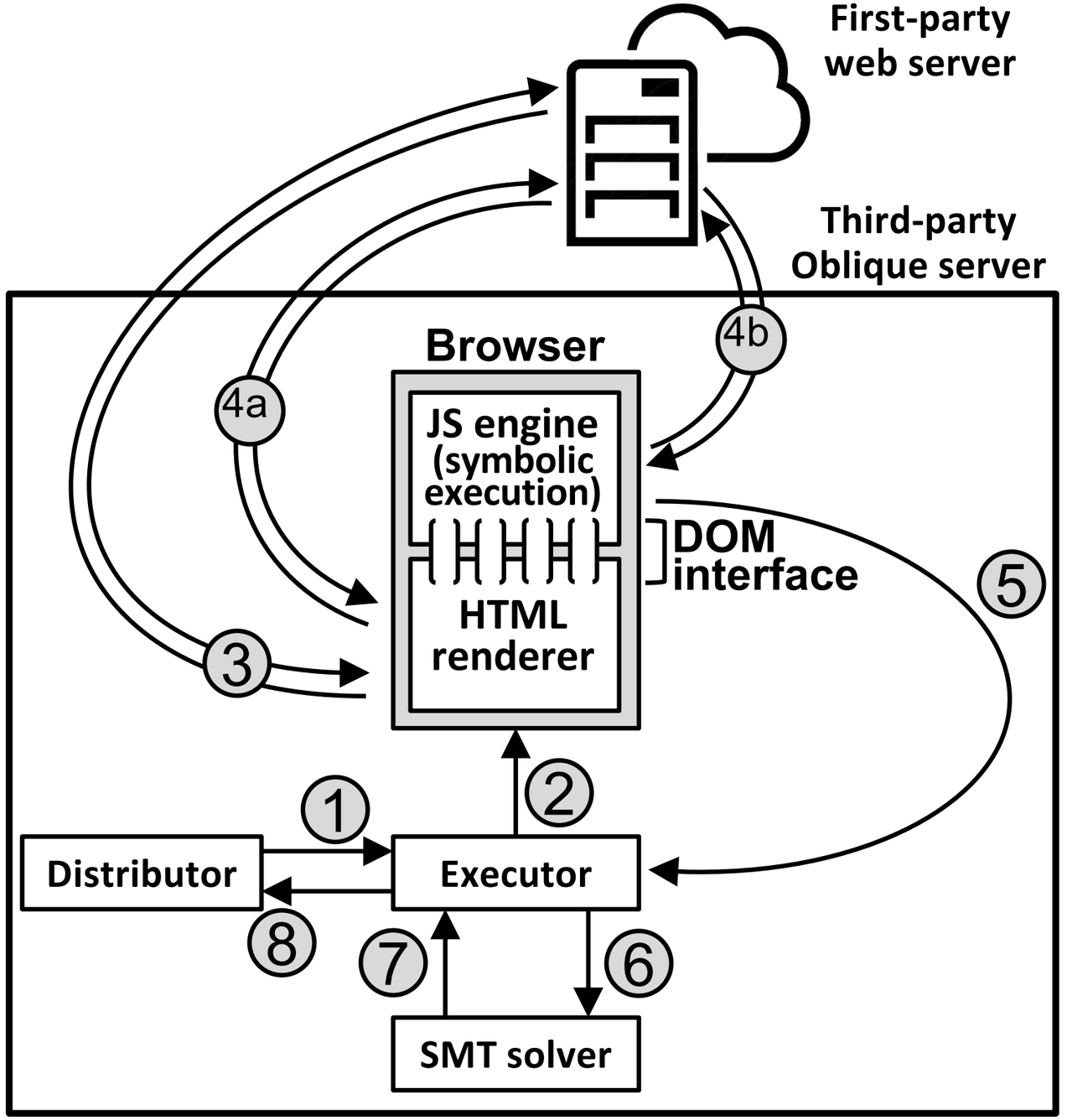
5. Once the page load finishes, the symbolic path constraint, e.g. `document.cookie = “cat”{{\w*}}` is sent to executor
6. Executor asks the SMT solver to invert part of the constraint
7. Solver performs inversion, e.g., `document.cookie = (^“cat”){{\w*}}` and concretizes the new constraint, e.g., `document.cookie = “x81b5”`
8. Solver returns the new test input to the [illegible] who inserts the input into a priority queue

This input would explore the new universe!



Symbolic Analysis

The longer we run the symbolic analysis, the more universes we discover!

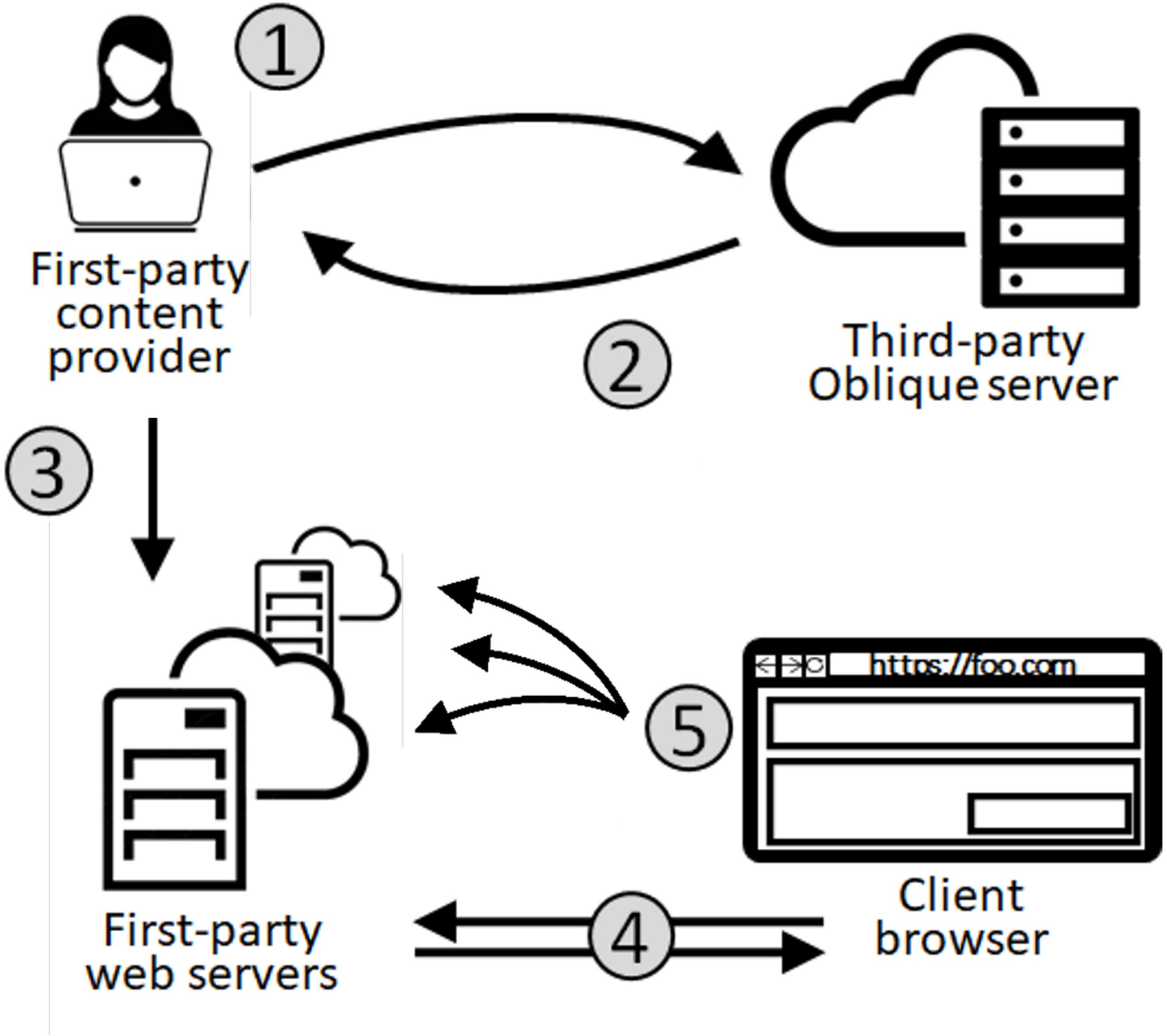


Input name	HTTP header	JavaScript variable	Description
User Agent	User-Agent	navigator.userAgent	The local browser type, e.g., "Mozilla/5.0 (Windows; U; Win98; en-US; rv:0.9.2) Gecko/20010725 Netscape6/6.1"
Platform	Included in User-Agent	navigator.platform	The local OS, e.g., "Win64"
Screen characteristics	N/A	window.screen.*	Information about the local display, e.g., the dimensions and pixel depth
Host	Host	location.host	Specifies the virtual host and port number to use
Referer	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 Referer, but only includes the origin part of the referring URL
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 text string containing "key=value" pairs

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

Overview

1. Developer uploads page content to Oblique's third-party analysis server
2. Oblique returns a path constraint tree for the page
3. Developer uploads page content to first-party web servers
4. Later, user loads the page
5. Oblique's JavaScript library concretizes path constraint tree, prefetches objects

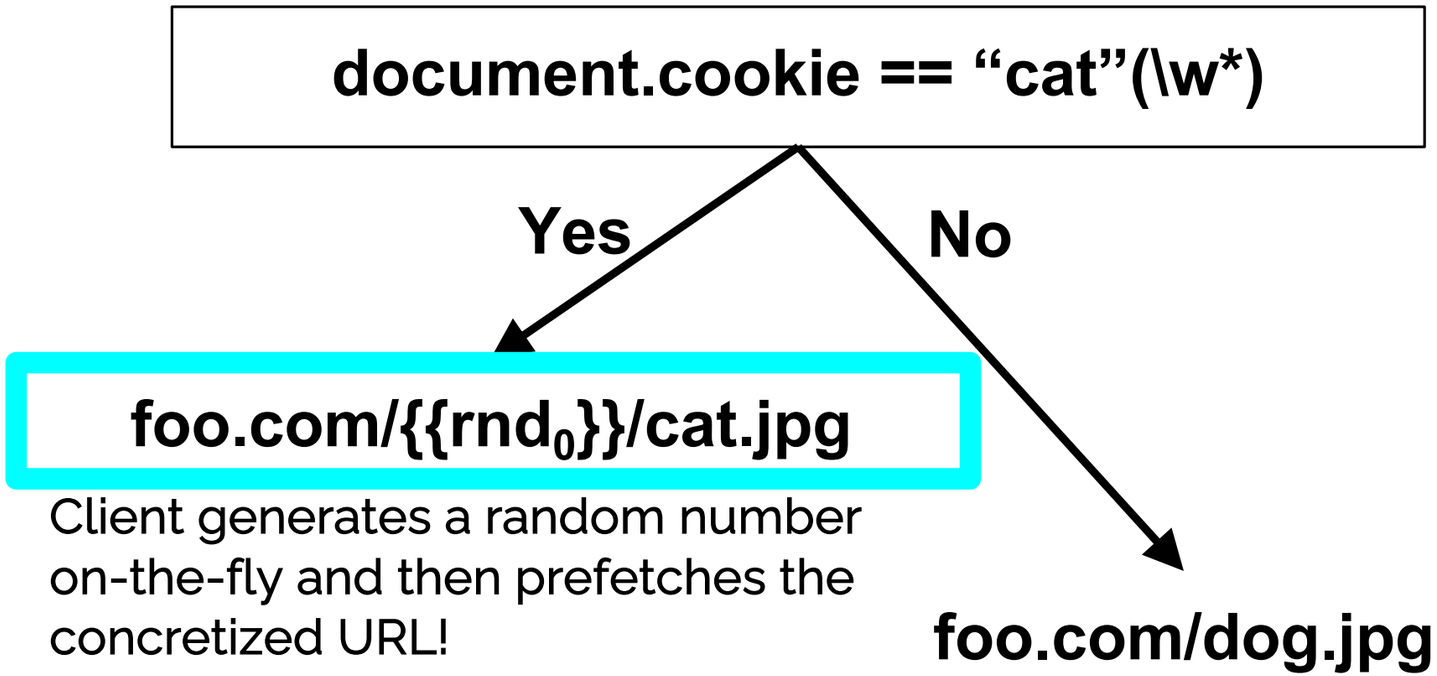


Overview

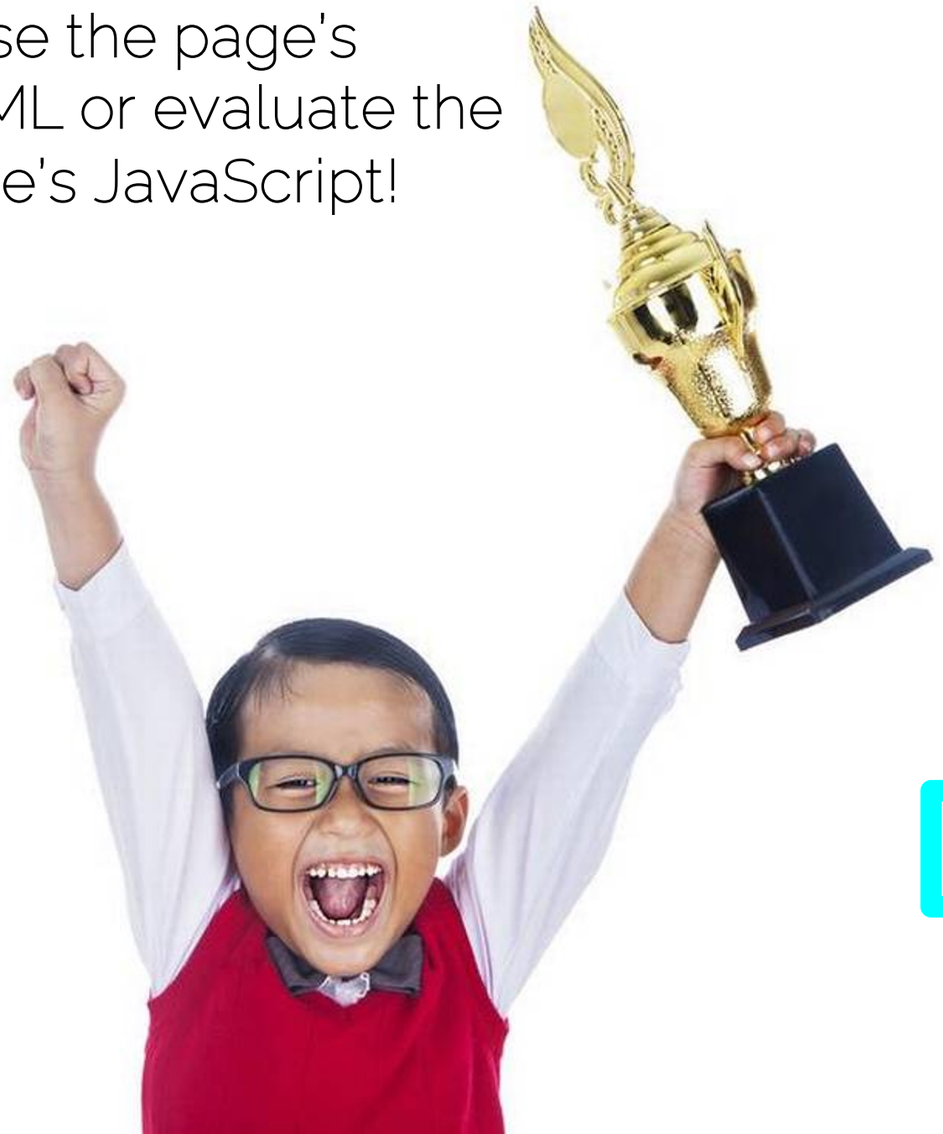
1. Developer uploads page content to Oblique's third-party analysis server
2. Oblique returns a path constraint tree for the page
3. Developer uploads page content to first-party web servers
4. Later, user loads the page
5. Oblique's JavaScript library concretizes path constraint tree, prefetches objects

```
var baseUrl = "foo.com/";
var rndId = Math.random().toString();
if(document.cookie.indexOf("cat")==0){
   fetch(baseUrl + rndId + "/cat.jpg");
}else{
   fetch(baseUrl + "/dog.jpg");
}
```

"cat=OfCourse; id=42"



Client determines which URLs to fetch without having to parse the page's HTML or evaluate the page's JavaScript!



“cat=OfCourse; id=42”

document.cookie == “cat”(\w*)

Yes

No

foo.com/{{rnd$_0$}}/cat.jpg

Client generates a random number on-the-fly and then prefetches the concretized URL!

foo.com/dog.jpg

LET’S BE HONEST WITH EACH OTHER

Program Analysis Techniques Somewhat Don't Work

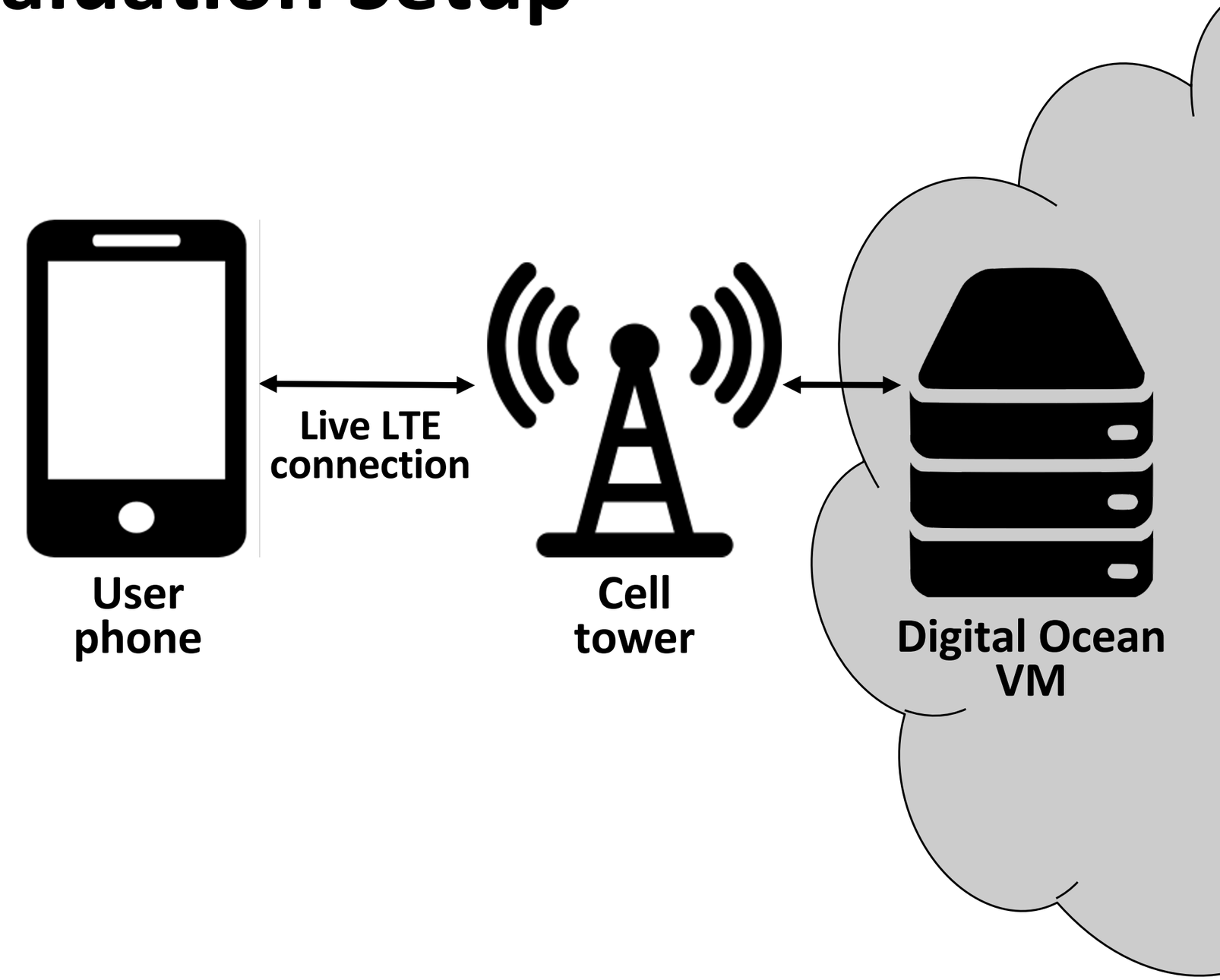
- Taint tracking
  - Implicit flows
  - XOR'ing something with itself should clear taint
  - More generally, some kinds of code (e.g., hash functions) should destroy taint BUT HOW DO YOU KNOW?
- Formal methods: AIN'T NOBODY GOT TIME TO WRITE BIG SYSTEMS IN A FORMAL LANGUAGE
- Symbolic analysis
  - You almost never have enough time
  - Constraint solvers can't solve all constraints
  - Black-box code can be hard to model

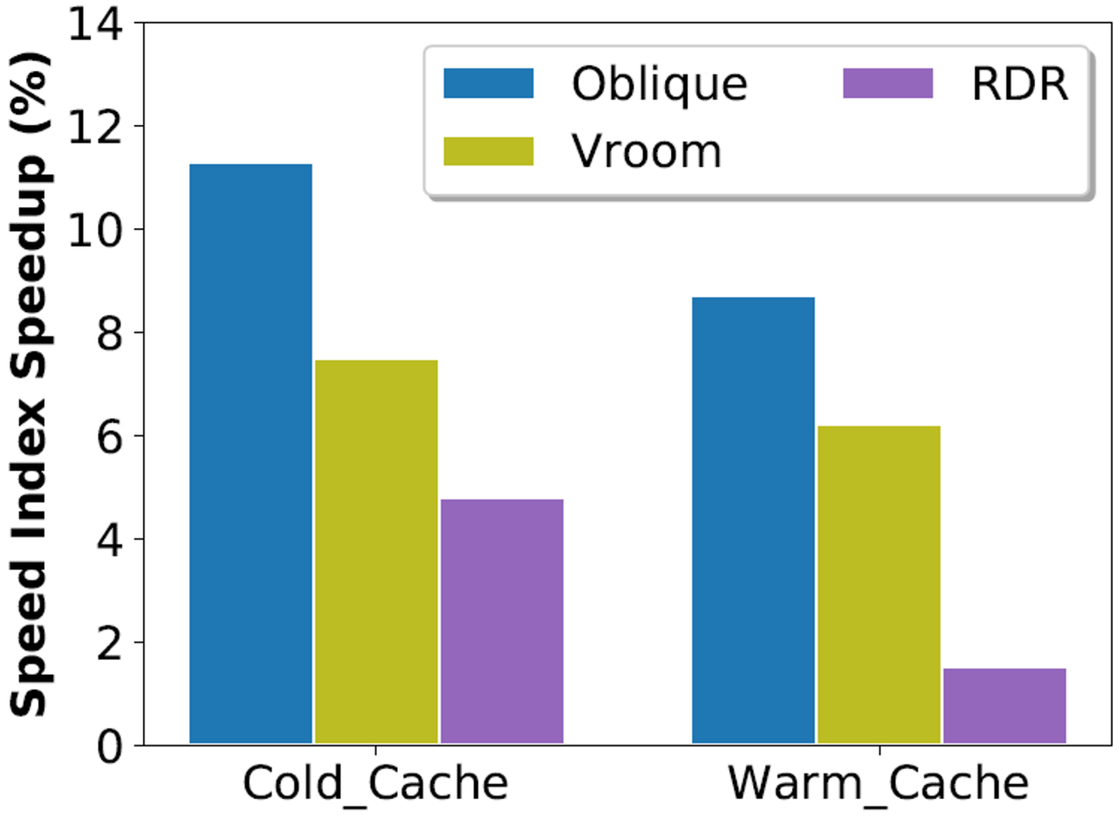
Limitations of Oblique

- Only certain native methods are modelled
  - For example, if the JavaScript string variable **s** contains symbolic data derived from **User-Agent**, then **String.charAt(s)**'s return value will properly capture that symbolic data
  - In contrast, Oblique always treats **Intl.DateTimeFormat(s)** as fully concrete, possibly hurting path coverage
- Oblique's concolic analysis may time out, hurting path coverage
- Oblique's concolic analysis can't issue HTTP requests that are nonidempotent

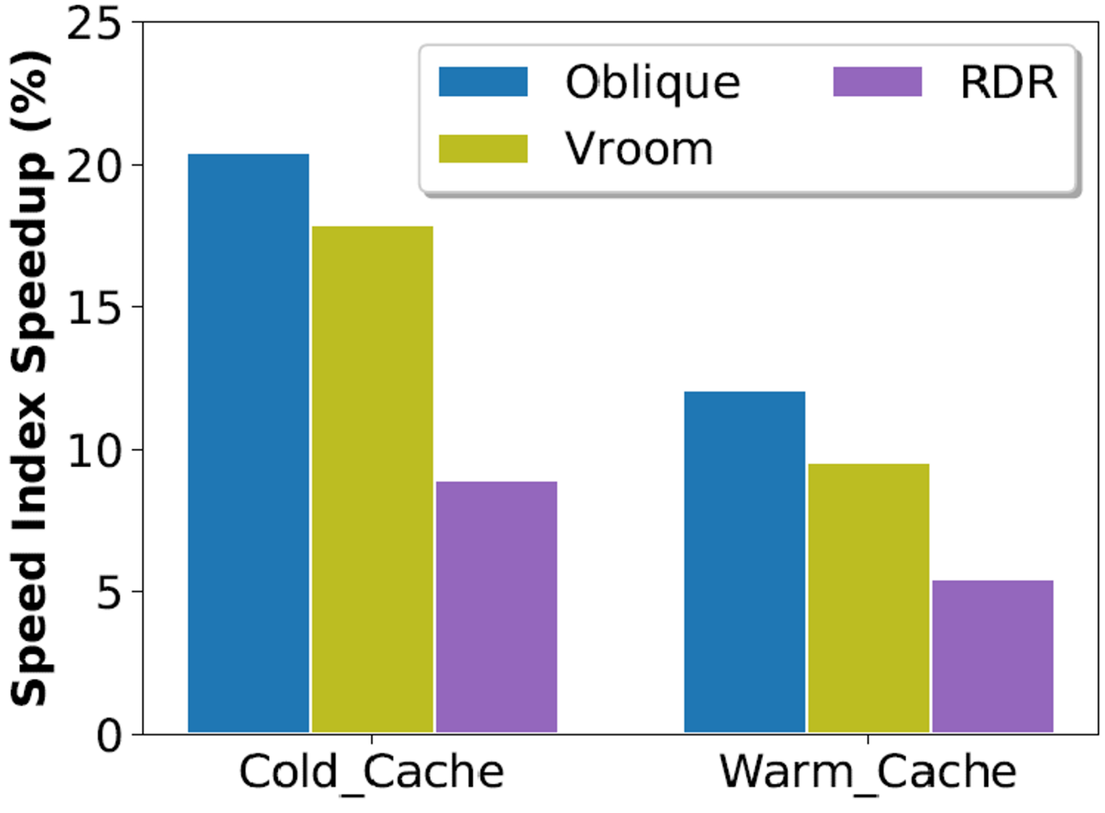
Evaluation Setup

- A Mahimahi derivative recorded content from 200 popular pages
- Digital Ocean VM ran:
  - Oblique web server
  - Vroom server
  - RDR server
- User device was a Galaxy S10e phone running Chromium v78
  - End-to-end RTT b/w phone and Digital Ocean VM was ~47ms
  - We used **netem** to inject added latency in some experiments

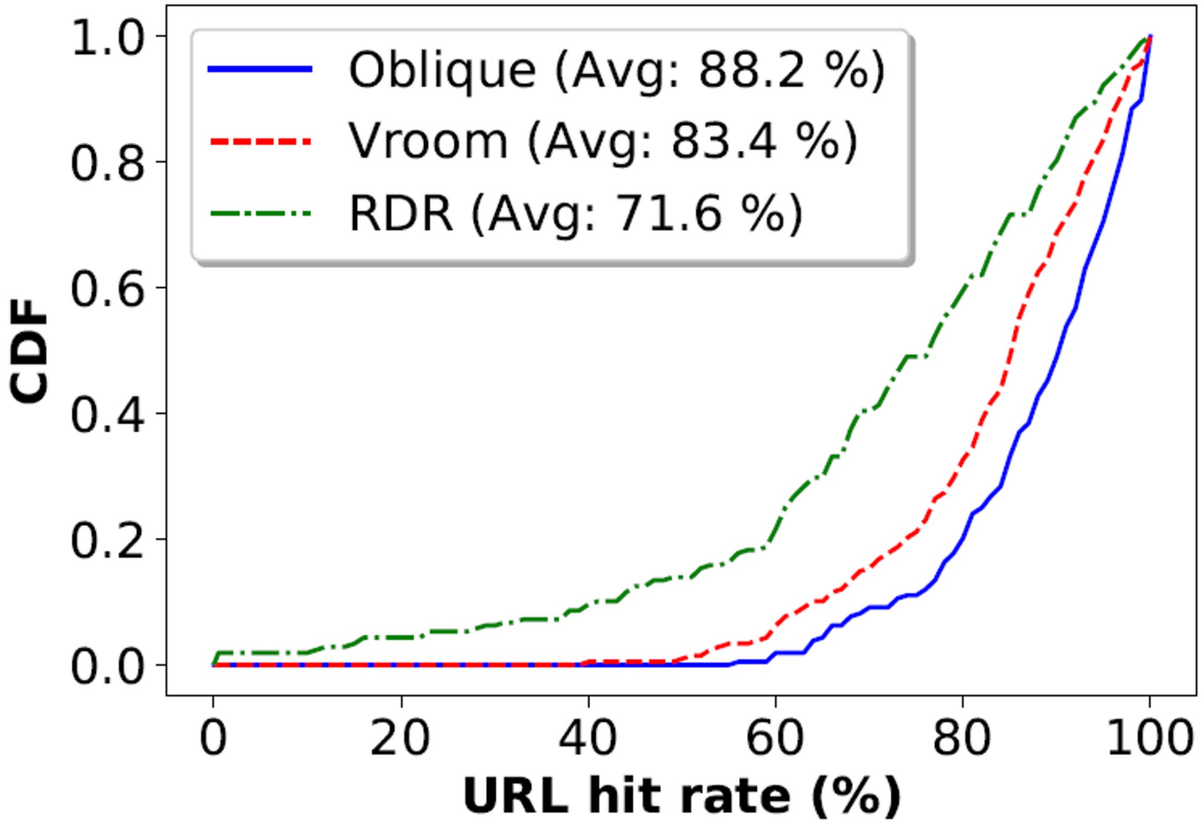




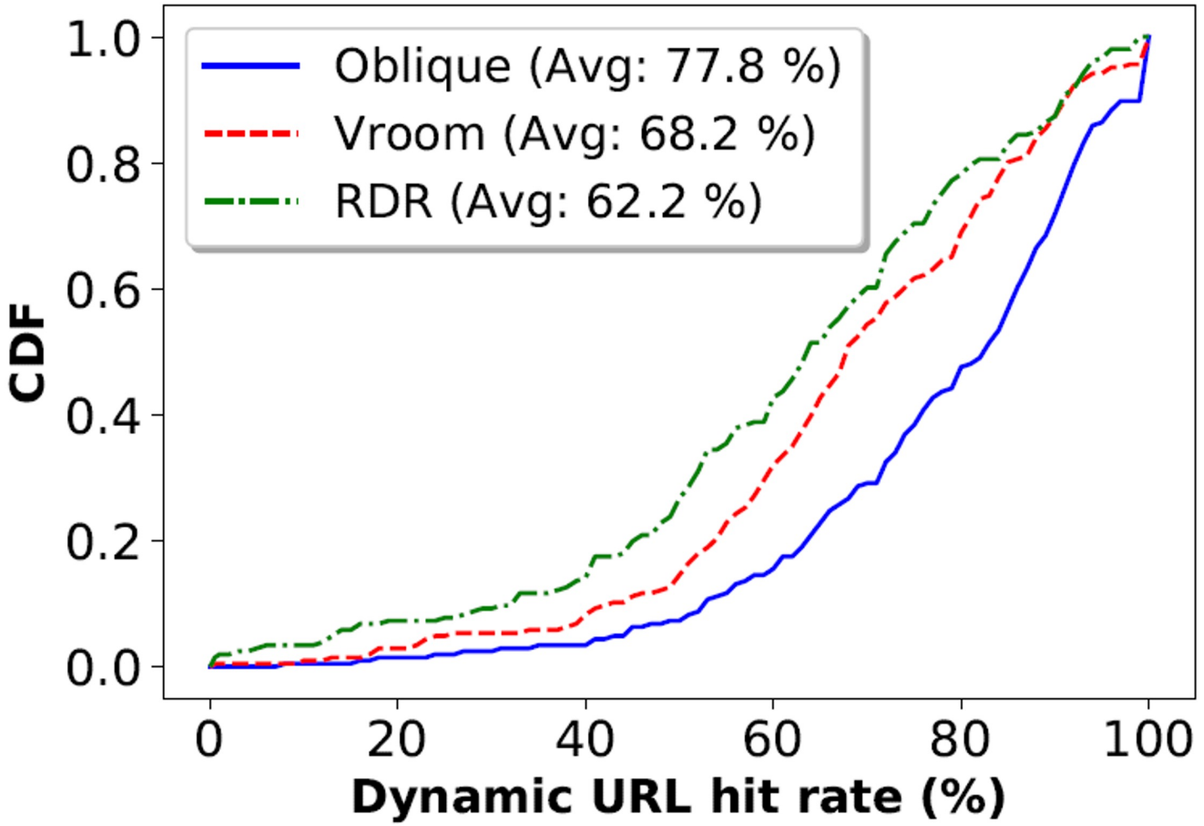
Speed Index (47 ms RTT)



Speed Index (150 ms RTT)



Prefetch hit rate (static+dynamic URLs)

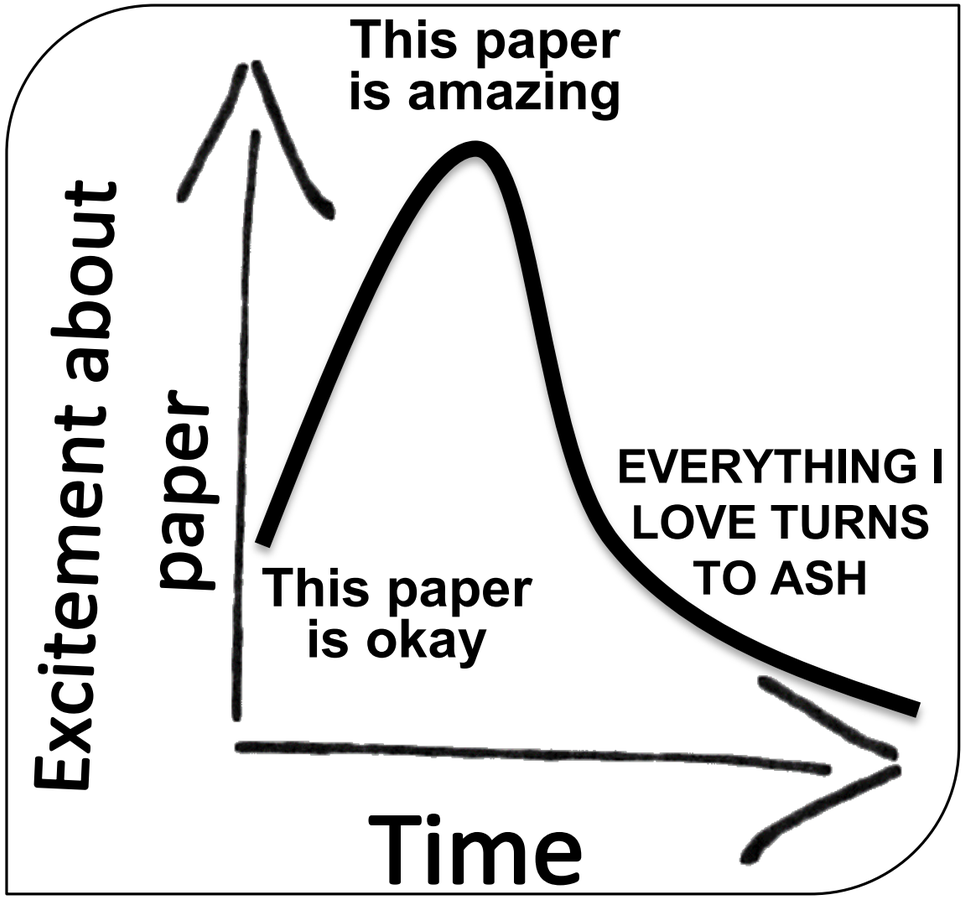


Prefetch hit rate (only dynamic URLs)

Outline

- How A Browser Loads a Web Page
- Impediments to Optimization
- Oblique
- What I Learned While Working On Oblique

Build a Supportive Community



The Atlantic

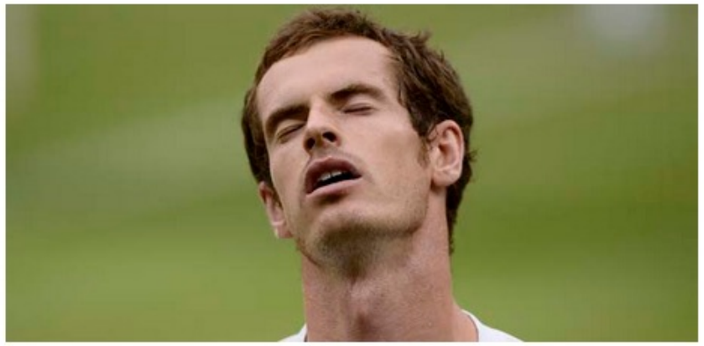
Why Losing Wimbledon Hurts So Much

By Kevin Craft

JULY 9, 2012

SHARE

Ask Andy Murray and Roger Federer: In tennis, the pain of defeat outweighs the thrill of victory.



Sports can seem like a cruel joke perpetrated on the athletes that entertain us week after week. This is because no athlete can escape the paradox at the heart of sports, which is that the agony of defeat always outweighs the thrill of victory. Listen to a great athlete articulate why he or she sacrifices so much and works so hard to compete at a high level, and more often you'll hear them say that a fear of losing is the chief source of motivation. Rarely if ever will an athlete say that a joy of winning is what keeps them going, and this careful choice of words is critical to understanding the psychology of sports. Winning may bring a certain level of satisfaction, but losing inspires a visceral feeling of pain, like a sharp punch to the gut, that can stay with someone long after competition is complete. And this means that even if an athlete wins half of the time over the course of a career, he or she will end up experiencing more painful emotions than joyful ones.

Tennis, more than any other professional sport, seems to amplify the pain of losing. When a tennis player loses a match, he or she must face the pain all alone. There are no teammates to turn to, no caddy to give an enthusiastic pat on the back as the match slips from one's grasp. A losing tennis player must shake hands, wave to the crowd and sit there alone, contemplating what could have been. [illegible]

In his book *Strokes of Genius*, an account of the epic 2008 Wimbledon final between Federer and Nadal, author Jon Wertheim quotes Toni Nadal, Rafa's uncle and coach, saying "Victory does not feel so good as losing feels bad. When you have a son, you are happy. But it's no comparison to the sadness you feel losing a son." While his metaphor may be a bit strong--losing a sporting event and losing a child are not commensurate in terms of grief--that quote underscores the truth about winning and losing in sports. [illegible]

In academia (and life in general), other people's failures are often invisible to you!

0 rejections

1 rejection

6 rejections

And then got the Best Paper award at SOCC 2019!

Polaris: Faster Page Loads Using Fine-grained Dependency Tracking

Ravi Netravali*, Ameesh Goyal*, James Mickens†, Hari Balakrishnan*
*MIT CSAIL †Harvard University

Abstract

To load a web page, a browser mu
uate objects like HTML files and
code. Evaluating an object can resul
jects being fetched and evaluated. Th
page requires a browser to resolve a
this partial ordering constrains the se
browser can process individual obje
many edges in a page's dependency g
able by today's browsers. To avoid vio
dependencies, browsers make conser
about which objects to process next, l
and CPU underutilized.

We provide two contributions. First
surement platform called Scout that
data flows across the JavaScript he
we show that prior, coarse-grained d
ers miss crucial edges: across a t
pages, prior approaches miss 30% o
dian, and 118% at the 95th percentile.
tify the benefits of exposing these
browsers. We introduce Polaris, a d
scheduler that is written in JavaScrip
modified browsers; using a fully a
servers can translate normal pages i
themselves with Polaris. Polaris uses
dency graphs to dynamically determi
load, and when. Since Polaris' graph
edges, Polaris can aggressively fetch o
minimizes network round trips. Exper
of network conditions show that Pola
load times by 34% at the median, an
percentile.

1 Introduction

Users demand that web pages load qu
of just a few milliseconds can result
ing a page early; such early abandon
lions of dollars in lost revenue for pag

Prophecy: Accelerating Mobile Page Loads Using Final-state Write Logs

Ravi Netravali*, James Mickens†
*MIT CSAIL, †Harvard University

Abstract

Web browsing on mobile devices is expensive in terms of battery drainage and bandwidth consumption. Mobile pages also frequently suffer from long load times due to high-latency cellular connections. In this paper, we introduce Prophecy, a new acceleration technology for mobile pages. Prophecy simultaneously reduces energy costs, bandwidth consumption, and page load times. In Prophecy, web servers precompute the JavaScript heap and the DOM tree for a page; when a mobile browser requests the page, the server returns a write log that contains a single write per JavaScript variable or DOM node. The mobile browser replays the writes to quickly reconstruct the final page state, eliding unnecessary intermediate computations. Prophecy's server-side component generates write logs by tracking low-level data flows between the JavaScript heap and the DOM. Using knowledge of these flows, Prophecy enables optimizations that are impossible for prior web accelerators; for example, Prophecy can generate write logs that interleave DOM construction and JavaScript heap construction, allowing interactive page elements to become functional immediately after they become visible to the mobile user. Experiments with real pages and real phones show that Prophecy reduces median page load time by 53%, energy expenditure by 36%, and bandwidth costs by 21%.

1 Introduction

Mobile browsing now generates more HTTP traffic than desktop browsing [18]. On a smartphone, 63% of user focus time, and 54% of overall CPU time, involves a web browser [56]; mobile browsing is particularly important in developing nations, where smartphones are often a user's sole access mechanism for web content [12, 23]. So, mobile page loads are important to optimize along multiple axes: bandwidth consumption, energy consumption, and page load time. Reducing bandwidth overhead allows users to browse more pages without violating data plan limits. Reducing energy con-

computes the JavaScript state and the DOM st
belongs to a loaded version of a frame. The p
puted JavaScript heap and DOM tree represent
of objects; however, one of Prophecy's key ins
that this state should be transmitted to clients
form of *write logs*, not serialized graphs. At a hig
a write log contains one write operation per
in the frame's load-time state. By returning wri
for each variable's final state, instead of returni
ditional, unprocessed HTML, CSS, and JavaScr
browser can elide slow, energy-intensive compu
involving JavaScript execution and graphical laye
dering. Conveniently, Prophecy's write logs for
are smaller than the frame's original content, and
fetched in a single HTTP-level RTT. Thus, Proj
precomputation also decreases bandwidth consu
and the number of round trips needed to build a f

Earlier attempts at applying precomputation
sites have suffered from significant practical lim
(§6), in part because these systems used serialized
instead of write logs. Serialized graphs hide dat
that write logs capture; analyzing these data f
necessary to perform many optimizations. For ex
Prepack [16] cannot handle DOM state, and is ur
elide computation for some kinds of common Jav
patterns. Shandian [51] does not support caching
majority of a page's content, does not support i
ate page interactivity (§3.5), and does not work
modified commodity browsers; furthermore, Sh
exposes all of a user's cookies to a single prox
ing significant privacy concerns. In contrast, Pr
works on commodity browsers, handles both DO
JavaScript state, preserves traditional same-origi
cies about cookie security, and supports byte-gra
caching (which is *better* than HTTP's standard fi
caching scheme). Prophecy can also prioritize th
ing of interactive state; this feature is important f
that load over high-latency links, and would otl
present users with rendered GUIs that may not

Vesper: Measuring Time-to-Interactivity for Web Pages

Ravi Netravali†*, Vikram Nathan†*, James Mickens‡, Hari Balakrishnan†
†MIT CSAIL, ‡Harvard University

Abstract

Everyone agrees that web pages should load more quickly. However, a good definition for "page load time" is elusive. We argue that, for pages that care about user interaction, load times should be defined with respect to *interactivity*: a page is "loaded" when above-the-fold content is visible, and the associated JavaScript event handling state is functional. We define a new load time metric, called *Ready Index*, which explicitly captures our proposed notion of load time. Defining the metric is straightforward, but actually measuring it is not, since web developers do not explicitly annotate the JavaScript state and the DOM elements which support interactivity. To solve this problem, we introduce Vesper, a tool that rewrites a page's JavaScript and HTML to automatically discover the page's interactive state. Armed with Vesper, we compare Ready Index to prior load time metrics like Speed Index; across a variety of network conditions, prior metrics underestimate or overestimate the true load time for a page by 24%–64%. We introduce a tool that optimizes a page for Ready Index, decreasing the median time to page interactivity by 29%–32%.

1 Introduction

Users want web pages to load quickly [31, 40, 42]. Thus, a vast array of techniques have been invented to decrease load times. For example, browser caches try to satisfy network requests using local storage. CDNs [9, 27, 36] push servers near clients, so that cache misses can be handled with minimal network latency. Cloud browsers [4, 29, 34, 38] resolve a page's dependency graph on a proxy that has low-latency links to web servers; this allows a client to download all objects in a page using a single HTTP round-trip to the proxy.

All of these approaches try to reduce page load time. However, an inconvenient truth remains: none of these techniques directly optimize the speed with which a page becomes *interactive*. Modern web pages have sophisticated, dynamic GUIs that contain both visual and pro-

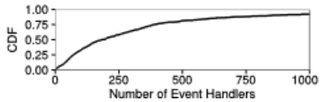


Figure 1: For the Alexa US Top 500 sites, we observe the median number of GUI event handlers to be 182.

for a page to be ready for user interaction. As shown Figure 1, pages often contain *hundreds* of event handle that drive interactivity.

In this paper, we propose a new definition for load tim that directly captures page interactivity. We define a pag to be fully loaded when:

(1) the visual content in the initial browser viewpor has completely rendered, and
(2) for each interactive element in the initial view port, the browser has fetched and evaluated th JavaScript and DOM state that supports the el ment's interactive functionality.

Prior definitions for page load time overdetermine or u derdetermine one or both of those conditions (§2), lea ing to inaccurate measurements of page interactivity. F example, the traditional definition of page load time, represented by the JavaScript `onload` event, captur when *all* of a page's HTML, JavaScript, CSS, and imag have been fetched and evaluated; however, this definitio is overly conservative, since only a subset of that sta may be needed to allow a user to interact with the conte in the initial viewport. Newer metrics like above-the-fo time [21] and Speed Index [14] measure the time that page needs to render the initial viewport. However, thes metrics do not capture whether the page has loaded cri ical JavaScript state (e.g., event handlers that respond GUI interactions, or timers that implement animations)

To accurately measure page interactivity, we must d termine when conditions (1) and (2) are satisfied. Dete mining when condition (1) has been satisfied is relative straightforward, since rendering progress can be me

Reverb: Speculative Debugging for Web Applications

Ravi Netravali
UCLA

James Mickens
Harvard University

ABSTRACT

Bugs are common in web pages. Unfortunately, traditional debugging primitives like breakpoints are crude tools for understanding the asynchronous, wide-area data flows that bind client-side JavaScript code and server-side application logic. In this paper, we describe Reverb, a powerful new debugger that makes data flows explicit and queryable. Reverb provides three novel features. First, Reverb tracks *precise value provenance*, allowing a developer to quickly identify the reads and writes to JavaScript state that affected a particular variable's value. Second, Reverb enables *speculative bug fix analysis*. A developer can replay a program to a certain point, change code or data in the program, and then resume the replay; Reverb uses the remaining log of nondeterministic events to influence the post-edit replay, allowing the developer to investigate whether the hypothesized bug fix would have helped the original execution run. Third, Reverb supports *wide-area debugging* for applications whose server-side components use event-driven architectures. By tracking the data flows between clients and servers, Reverb enables speculative replaying of the *distributed* application.

KEYWORDS

record-and-replay debugging, systems debugging

ACM Reference Format:
Ravi Netravali and James Mickens. 2019. Reverb: Speculative Debugging for Web Applications. In *SoCC '19: ACM Symposium on Cloud Computing conference, Nov 20–23, 2019, Santa Cruz, CA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3357223.3362733

1 INTRODUCTION

Debugging the client-side of a web application is hard. The DOM interface [40], which specifies how JavaScript code interacts with the rest of the browser, is sprawling and constantly accumulating new features [27, 35]. Furthermore, the DOM interface is pervasively asynchronous and event-driven, making it challenging for developers to track causality across event handlers [26, 36, 49, 72]. As a result, JavaScript bugs are endemic, even on popular sites that are maintained by professional developers [57, 59].

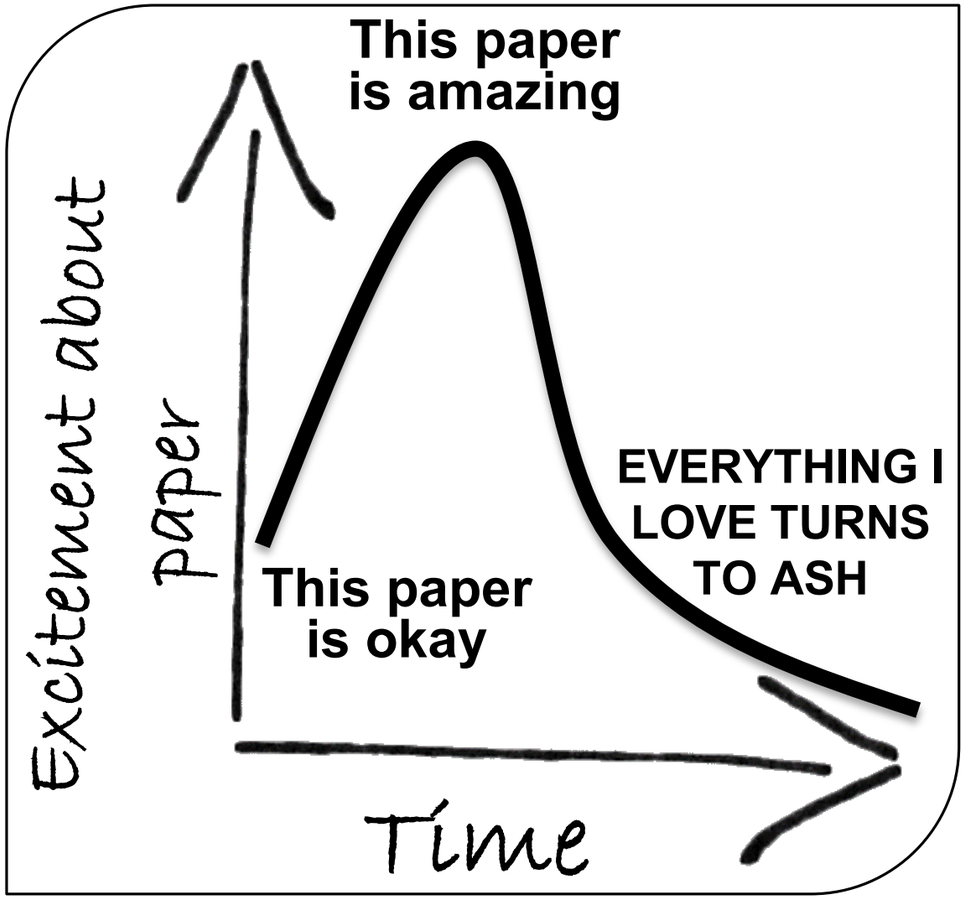
Commodity browsers include JavaScript debuggers that support breakpoints and watchpoints. However, fixing bugs is still hard. Breakpoints and watchpoints let developers inspect program state

at a moment in time; however, in an event-driven program with extensive network and GUI interactions, bug diagnosis often requires complex temporal reasoning to reconstruct a buggy value's *provenance* across multiple asynchronous code paths. This provenance data is not exposed by more advanced tools for replay debugging or program slicing (§2).

In this paper, we introduce Reverb, a new debugger for web applications. Reverb has three features which enable a fundamentally more powerful debugging experience. First, Reverb tracks *precise value provenance*, i.e., the exact set of reads and writes (and the associated source code lines) that produce each program value. Like a traditional replay debugger [13, 36, 62], Reverb records all of the nondeterministic events from a program's execution, allowing Reverb to replay a buggy execution with perfect fidelity. Unlike a traditional replay debugger, Reverb also records the *deterministic* values that are manipulated by reads and writes of page state. Using this extra information at replay time, Reverb enables developers to query fine-grained data flow logs and quickly answer questions like "Did variable `x` influence variable `y`?" or "Was variable `z`'s value affected by a control flow that traversed function `f()`?" Reverb's logging of both nondeterministic and deterministic events is fast enough to run in production: for the median web page in our 300 page test corpus, Reverb increases page load time by only 5.5%, while producing logs that are only 45.4 KB in size.

Reverb's second unique feature is support for *speculative bug fix analysis*. At replay time, Reverb allows a developer to pause the application being debugged, edit the code or data of the application, and then resume the replay. Post-edit, Reverb replays the remaining nondeterministic events in the log, using carefully-defined semantics (§3.3) to determine how those events should be replayed in the context of the edited program execution. Once the altered execution has finished replaying, Reverb identifies the control flows and data flows which differ in the edited and original executions. These analyses help developers to determine whether a hypothesized bug fix would have helped the original program execution. Speculative edit-and-replay is unsound, in the sense that a post-edit program can misbehave in arbitrary ways, e.g., by attempting to read an undefined variable. However, even without Reverb, *the process of testing bug fixes is unsound*. A developer typically lacks a priori knowledge about whether a hypothesized fix will work. The developer implements the hypothesized fix, and then runs tests and tries to determine whether the fix actually worked: even if all of the tests pass, there is

Build a Supportive Community



- You are not the only one who:
  - Has papers get rejected
  - Occasionally says something incorrect during a meeting
  - Isn't sure which career path is the best one
- A strong support group is important
  - Talk to other students (even in other departments!)
  - Make time to not do work
  - Don't be afraid to talk to mental health professionals

Conclusion

- Prefetching helps to reduce page load times
- Prior systems generate prefetch lists by:
  - Breaking TLS integrity, or
  - Preventing third-party outsourcing of the analysis
- Oblique uses symbolic execution to eliminate the design tension
  - Oblique's third-party server can model user-specific data as symbols
  - Symbols are only resolved by clients!
- Oblique reduces page loads by up to 31%, outperforming Vroom and RDR by up to 17%