Fathom: A Browser-based Network Measurement Platform

Mohan Dhawan[*]
Rutgers University
mdhawan@cs.rutgers.edu

Justin Samuel
UC Berkeley
jsamuel@cs.berkeley.edu

Renata Teixeira[✿]
CNRS & UPMC
renata.teixeira@lip6.fr

Christian Kreibich
ICSI & UC San Diego
christian@icir.org

Mark Allman
ICSI
mallman@icir.org

Nicholas Weaver
ICSI & UC San Diego
nweaver@icsi.berkeley.edu

Vern Paxson
ICSI & UC Berkeley
vern@cs.berkeley.edu

ABSTRACT

For analyzing network performance issues, there can be great utility in having the capability to measure directly from the perspective of end systems. Because end systems do not provide any external programming interface to measurement functionality, obtaining this capability today generally requires installing a custom executable on the system, which can prove prohibitively expensive. In this work we leverage the ubiquity of web browsers to demonstrate the possibilities of browsers themselves offering such a programmable environment. We present *Fathom*, a Firefox extension that implements a number of measurement primitives that enable websites or other parties to program network measurements using JavaScript. Fathom is lightweight, imposing $< 3.2\%$ overhead in page load times for popular web pages, and often provides 1 ms timestamp accuracy. We demonstrate Fathom's utility with three case studies: providing a JavaScript version of the *Netalyzr* network characterization tool, debugging web access failures, and enabling web sites to diagnose performance problems of their clients.

Categories and Subject Descriptors

C.2.0 [**Computer-Communication Networks**]: General; C.4 [**Performance of Systems**]: Measurement techniques; D.2.8 [**Software Engineering**]: Metrics—*performance measures*

General Terms

Design, Measurement, Performance, Security

Keywords

Network troubleshooting, network performance, end-host network measurement, web browser, browser extension

---

[*]This work was done while the author was visiting ICSI.

[✿]This work was done while the author was visiting UC Berkeley & ICSI.

1 Introduction

Network measurements from the perspective of end systems often play a vital role both for researchers who want to develop general insight into how the Internet functions, and for practitioners aiming to diagnose individual performance issues. To gain insight into network operation, researchers traditionally either (i) beg acquaintances to install a custom measurement tool, or (ii) use or instantiate infrastructure to conduct the measurements (e.g., PlanetLab). Neither of these options scales to supporting broad measurement of the network's edge, limiting the scope of the insight we can obtain. Practitioners, on the other hand, have less interest in developing general knowledge about networked systems and more interest in understanding the performance of their own systems, including understanding *why* their users experience suboptimal performance. While systems developed by the research community for general measurement may aid practitioners in small ways, these systems do not run on regular users' end systems and hence do not offer help with *specific* measurement or debugging of user problems.

While researchers and practitioners come to the need for measurement of user end systems from opposite ends of the spectrum—i.e., one looking for broad, general results, and the other seeking specific information—both find obtaining such measurements problematic due to the lack of general support in end systems. While getting users to install programs can suffice for particular, limited situations, the approach does not scale up: one cannot rely upon users repeatedly conducting such installations as a building block for facilitating measurement. One difficulty in this regard concerns users not wanting to continually undergo the hassle of the process. Another difficulty is the challenge that implementors face in developing portable executables that run correctly across a wide range of system configurations. A third issue is one of safety: such installations, whether for standalone binaries or native code that extends an existing system, generally require privileged access and trust in the behavior of the introduced program.

In this work we aim to overcome these difficulties by minimizing the impediments to writing and installing measurement programs across a large number of end systems. Our vehicle for doing so is *Fathom,* a browser-based extensible measurement platform. Fathom provides a measurement-oriented API that standard JavaScript loaded from any web page can tap into to facilitate empirical observation.

We argue that building a measurement platform into the web browser hits a sweet spot for both deployability and flexibility, thus providing Fathom with the potential to flourish where other mea-

surement systems have floundered. The ubiquity of web browsers, and the central role they play for a vast array of disparate network activities, provide an unparalleled scaling potential that holds great promise for benefiting common measurement and troubleshooting scenarios (§ 2). Like other measurement platforms, Fathom requires users to install a piece of software, a step that generally presents a significant hurdle to widespread adoption. However, the user only has to do so *once* to enable a vast range of measurements. By bundling useful Fathom applications with the platform itself (most notably, a connectivity debugger that works even in complete absence of connectivity), we provide users an immediate incentive for installation.

We begin by framing desiderata for a browser-based measurement platform (§ 3), which guide both Fathom's design and its implementation. Our design (§ 4) includes a set of APIs covering a wide range of measurement and troubleshooting applications, significantly expanding the capabilities of in-page JavaScript code. To ensure safety, we equip Fathom with a security model that balances user approval, permission management, and code-signing (§ 4.2 and § 4.3). Consideration of possible implementation strategies motivates our decision to implement Fathom as a JavaScript-only extension for the Firefox browser, paying close attention to the constraints imposed by modern-day browser environments (§ 5). We demonstrate that careful design of a JavaScript-driven measurement platform enables experimenters to conduct measurements with modest overhead for the user (§ 6.1) and high accuracy (§ 6.2). We report on additional experiences gained using Fathom in three different use cases (§ 7). We discuss portability and adoption incentives (§ 8), and close with a survey of related browser extension work and measurement platforms (§ 9).

Contributions: Fathom explores the feasibility of building a generic network measurement and troubleshooting platform into modern browsers. We view our contribution as developing a realization of broad capability that the community has been working towards for a number of years. Fathom builds on the lessons of previous efforts and reflects a promising, practical approach that (*i*) fosters innovation via open-endedness, as opposed to implementing a particular tool; (*ii*) provides unparalleled access to the edge via the ubiquity of browsers; (*iii*) enables novel analyses supported via passive collection of context and active experimentation; (*iv*) combines existing security primitives in a novel way to safely expand in-page JavaScript capabilities.

Fathom is open source and we invite interested readers to visit http://fathom.icsi.berkeley.edu to download the platform and provide feedback.

2 Motivating Scenarios

We begin by sketching examples of measurement efforts that benefit from both an end system vantage point and a large footprint. The list here is far from exhaustive; our design is general, and hence adaptable to a range of other tasks, too.

Web performance debugging: With a browser-based vantage point website operators can significantly improve their ability to diagnose slow load times. Along those lines, Google's analytics code recently started collecting basic page rendering times [24]. This captures facts, but cannot provide troubleshooting assistance, for which page authors currently lack critical information. The web pages' isolation from the host system here works against the developer, preventing access to helpful indicators such as system load and I/O characteristics. For example, the Google Maps forum [19] shows discussions whereby Google engineers explicitly ask users to quantify the "slowness" of page loads, including ancillary system-level information such as CPU and network activity.

By allowing Google engineers access to a measurement platform in the user's browser they could directly extract the required information. Researchers can also leverage this capability to better understand web performance and complexity (e.g., [7, 10]) both at scale and across sites.

Web access failures: Similar to the performance debugging case, a browser-based platform can also directly assist users in diagnosing website failures. In this scenario, the browser could provide a "diagnose" button that launches measurements to identify common causes of access failures (e.g., loss of connectivity or DNS issues). While web failures have been studied in a passive fashion [1, 42], giving users a way to conduct real-time assessment offers the potential to both (*i*) better diagnose the problem with contemporaneous active measurements and (*ii*) inform users what might be done to fix the problem (or at least who to ask to fix the problem).

Local network debugging: Poor performance can also stem from issues in the local network and hence combining external- and local-network measurements can help pinpoint the location of the problem [10]. Further, when multiple local systems use the measurement platform we can discover this—e.g., using multicast DNS or UPnP—and coordinate the vantage points for a richer analysis of the failure.

ISP performance: A wide range of tools perform active measurements from end systems to a controlled server, including speed tests [4], general performance diagnostics [20, 28, 31], traffic tampering [46], and traffic shaping detectors [12, 25]. While each of these tools performs custom tests, they share the basic need to conduct active probing from end systems.

Large-scale topology mapping: Many efforts within the research community over the years have tried to understand the Internet's topology by conducting traceroute(-like) measurements from multiple vantage points [8, 44, 47]. A browser-based platform would facilitate such studies from a breadth of vantage points currently not practical.

3 Platform Design Goals

We identify the broad design goals for an end-system based measurement platform as follows:

Rich measurement capabilities: The examples in § 2 show that we require support for both *passive* and *active* observations. Passive measurements provide precise and continuous understanding of the activity in the browser, ranging from system-level information such as CPU load, memory availability, and traffic load, to browser-level activity such as HTTP session tracking and fine-grained page-level activity (e.g., DOM[1] updates and event loop activity). Note that this enables far broader analyses than available to in-browser JavaScript debuggers (e.g., Firebug). While network problems may often manifest in passive measurements, their root cause frequently reveals itself only using active probing. We therefore require a wide range of network I/O primitives, reaching as far down in the networking stack as feasible.

Extensibility: As sketched in § 2, we strive for a general measurement facility rather than providing a simple interface to ready-made measurements. Therefore, instead of providing APIs for specific measurements, the platform should provide APIs that support the *implementation* of as wide an array of measurements as possible.

Security and privacy: While Fathom strives to maximize utility to practitioners and experimenters, the platform must not compromise the privacy of the user (e.g., expose browsing history). Addition-

---

[1] Document Object Model, a browser-independent tree abstraction representing a web page.

```
(1)   var fathom = window.fathom;
(2)   var DATA = "Hello world";
(3)   var ADDR = "226.0.0.2", PORT = 23456;

(4)   fathom.socket.multicast.open(socketCB);

(5)   function socketCB(sock) {
(6)     if (sock && !sock['error']) {
(7)       fathom.socket.multicast.send(sendCB,
                sock, DATA, ADDR, PORT);
(8)     }
(9)   }

(10)  function sendCB(result) { ...  }
```

Figure 1: Asynchronous multicast network output in Fathom.

ally, the platform must not allow for the abuse of host or network resources.

Measurement accuracy: We require the measurement platform to accurately track time and the outcome of network I/O events. Precise timing supports the ability to both understand when an event happened and to schedule future events.

Low system overhead: We desire the platform's runtime overhead to be as low as possible for two reasons: (i) high overhead can alter test results for intensive experiments (e.g., bandwidth measurements) and (ii) the platform resides in a user's application and hence we do not want to unduly impact the user's browsing activity.

Portability: Measurement code should work portably on any implementation of the platform. This implies that measurement code should execute without requiring compilation to different execution environments.

Fathom's design (§ 4) addresses the first three goals and its implementation (§ 5) the remaining goals.

4 Fathom

We design Fathom to provide a rich set of APIs for measurements. Since Fathom significantly expands the capabilities of in-page JavaScript code, it requires a carefully thought-out security and privacy model.

4.1 API design

Fathom's APIs remain conceptually independent of the underlying platform implementation and cover a broad range of network I/O capabilities as well as means to obtain system information and historical performance baselines. Our case studies (§ 7) and development experience drove the design.

To make its services available to in-page JavaScript code, Fathom creates a new JavaScript object, registers its APIs as methods on this object as requested by the measurement code (see § 4.3), and adds it to the page's `window` object as `window.fathom`. Web application code can invoke these APIs to perform both passive and active measurements on the end system. To do so, the caller provides a callback function which Fathom will invoke when the result of the call becomes available. This approach provides natural asynchronicity in the presence of potentially blocking network I/O. Fathom passes a single object to the callbacks. A set of properties on the object conveys error codes and return values. Figure 1 shows an example of sending data over a multicast socket. Fathom currently provides four main API families (listed in Table 1):

`fathom.socket` provides socket management. These functions export low-level socket functionality and serve as the basis for active measurements, including opening TCP and UDP sockets, setting socket options, and performing both blocking and non-blocking I/O operations.

`fathom.proto` supplies clients and servers for common application-layer protocols including DNS, multicast DNS, HTTP, and UPnP.

`fathom.system` realizes controlled access to system-provided tools and information to invoke select services on the host that can provide information otherwise unavailable or inconvenient to re-implement. Fathom sanitizes the call arguments, handles onerous output parsing, and returns results to the caller while abstracting platform specifics. Examples in this family include the host's *ping* and *traceroute* implementations. In contrast to other runtime plugins such as Java (see § 5.1), Fathom's API *never* allows invocation of arbitrary system commands.

Lastly, `fathom.utils` collects additional helpful APIs, particularly access to aggregate statistics for passively collected data (`metrics.*`) and configuration settings for the browser (`browser.*`), such as proxy configuration.

`fathom.utils.metrics.*` performs low-level system sampling and monitors the browser's HTTP activity. Fathom currently conducts passive data collection in three domains: each web page, the browser itself, and the host's system and network resource usage. For web pages, collection includes socket connection times, DNS resolution times, HTTP response times and goodput, as well as the trigger times for `DOMContentLoaded` and `Load` page events.[2] From the browser, it collects its overall memory use as well as network transmission statistics. Finally, from the host system it collects CPU load, memory use and availability, per-process load, and network interface statistics including packet/byte counters, transmission errors, and—for wireless links—signal strength and noise levels. Per-page CPU load proves tricky to measure. Fathom can expose any browser specific performance measurement API [37] to web pages to measure JavaScript overhead, but authors would still need to instrument their code for reliable profiling. Furthermore, the CPU load induced by active embedded objects such as Flash manifests outside of the browser.

Fathom keeps a history of these metrics to provide *baseline* performance metrics for the pages. JavaScript code using Fathom can then obtain aggregate statistics of these data points in order to diagnose detected anomalies in the document loading process. Fathom stores browser and system metrics in client-side persistent storage, while the web page metrics end up in the site's HTML5 `localStorage` facility. This separation of storage for the baseline metrics enables websites to selectively retain or discard collected datasets for individual browsing sessions, reduces Fathom's persistent storage requirements, and minimizes information leakage across different origins.

Since Fathom monitors the browser's entire HTTP activity, web pages not requiring Fathom's capabilities will also incur an overhead in page load times. To exclude any added overhead, we propose that sites that require Fathom's diagnostic capabilities could explicitly request them. To signal to the server that the client supports Fathom, Fathom adds an *X-Fathom* header in the HTTP request for each page. Web sites that wish to enable diagnostic measurements for the page and its additional resources echo this header in their response. Via local browser configuration the user can request passive data collection for sites as needed.

Comparison: Flash and Java constitute the main competitors to Fathom, so it behooves us to compare their capabilities directly. Fathom's network I/O capabilities subsume Flash's, with the exception of the latter's built-in multimedia streaming capabilities.

---

[2]When downloading a page's DOM has finished, the browser triggers a `DOMContentLoaded` event. When the page's additional components (such as images) have finished downloading as well, the browser triggers a `Load` event.

API	Availability in			Default policy (§ 4.3)
	JavaScript	Flash	Java Applet[1]	
`fathom.socket.tcp.*`	◐	◐[2]	●	U[3] / SM→U
`fathom.socket.udp.*`	○	○	●	U[3] / SM→U
`fathom.socket.broadcast.*`	○	○	●	U
`fathom.socket.multicast.*`	○	◐	●	U
`fathom.proto.dns.*`	○	○	●[4]	–
`fathom.proto.http.*`	◐	●	●	SM→U
`fathom.proto.mdns.*`	○	○	●[4]	U
`fathom.proto.upnp.*`	○	○	●[4]	U
`fathom.system.getActiveInterfaces()`	○	○	●	U
`fathom.system.getResolvers()`	○	○	●	U
`fathom.system.getRoutingTable()`	○	○	◐[5]	U
`fathom.system.doTraceroute()`	○	○	◐[5]	–
`fathom.system.doPing()`	○	○	◐[5]	–
`fathom.utils.browser.*`	◐[6]	◐[6]	◐[6]	U
`fathom.utils.timer.*`	●	●	●	–
`fathom.utils.metrics.*`	○	○	○	U
JavaScript & DOM access	●	◐[7]	◐[8]	–

[1] Given signed code and trust confirmed by the user, i.e., unconstrained host access with user-level privileges.

[2] No support for listening sockets.

[3] For listening sockets.

[4] If implemented by the experimenter.

[5] By directly calling system binaries in OS-specific locations.

[6] Implicitly or partially; for example, XML-HttpRequest connections transparently inherit browser proxy settings. In Flash, socket connections do not inherit, others do.

[7] By abusing the `getURL` function; no access to return values.

[8] JavaScript–Java interaction via LiveConnect; details depend on platform and implementation.

Table 1: Primary Fathom namespaces and APIs. Due to lack of space, we show select individual functions only for the `system` family of calls, given its diversity. The final column indicates input mechanisms required by the default client policy (see § 4.3): "U" symbolizes that a capability requires user input, "SM→U" means that server manifests suffice to permit the capability while their absence necessitates user input, and "–" indicates permission without additional confirmation. Fathom by design provides full access to JavaScript functions and the DOM without the need of an additional API; we include the ability here to compare to other platforms.

In particular, Fathom supports listening sockets, UDP, protocol-specific APIs, as well as controlled access to host commands and historical performance baselines. With Java, the situation depends on the execution context: without host-specific customization, untrusted applets only allow a limited set of network I/O APIs and require communication with the applet's origin server. As pointed out in § 5.1, signed and trusted applets (as well as Java applications) provide an over-privileged environment that supports all of Fathom's capabilities, without abstracting from platform specifics and without reliable DOM interaction. Table 1 summarizes the respective capabilities.

4.2 Privacy

Our first goal is to ensure Fathom does not allow measurements to exfiltrate sensitive information from a user's browser or host. To do so, Fathom's design mandates that its network I/O stack operates independently of the browser's stack. Hence, any state the browser would normally associate with a connection is unavailable to Fathom. For example, when Fathom script code running in a web page from site `a.com` issues HTTP connections to site `b.com`, such connections do not transmit current session cookies for `b.com` even though the user might currently be logged in to that site in a separate tab. This mechanism effectively defeats classic attacks leveraging same-origin violations, such as cross-site scripting (XSS) or cross-site request forgery (CSRF). Like any JavaScript code, Fathom scripts can only access host-local state (including cookies) for the site the script originated from (`a.com`, in this case).

By providing websites with the ability to collect additional client-side data in the form of both active and passive measurements, Fathom increases the ability for websites to fingerprint visitors [15]. In addition, clients using anonymizing proxies expose themselves to additional de-anonymization risks due to web pages having access to TCP and UDP sockets that can bypass the browser's proxy settings, as well as communicating with local network services (such as the local DNS resolver) that may expose identifying information. Current tools for obtaining anonymity or resisting fingerprinting try to automate disabling or changing sensitive browser functionality [45], but require vigilance on the user's part. Thus, it seems reasonable to expect developers of such technologies (or users employing it) to consider disabling Fathom as needed. Additionally, when the browser has a proxy setup, Fathom's default policy (see below) could disallow active measurement except through the proxy or by explicit agreement of the user. Doing so would also reduce the chances of fingerprinting through arbitrary active communication.

The storage of local data about a user's browsing history is another important privacy concern. Most popular browsers have adopted private browsing modes that, when enabled, aim to eliminate the local storage of information that could later reveal a user's browsing history [2]. Additionally, users expect that "clearing their history" removes all data the browser keeps that one could use to infer their history—behavior that Fathom conforms with.

4.3 Security

We next discuss Fathom's model for controlling scripts such that the platform has adequate protection against facilitating active attacks. We note that in the limit any communication we allow can become part of a distributed attack, but the same applies to browsers without Fathom [29, 50]. We strive to provide enough control that Fathom does not exacerbate the situation significantly. In our present design, Fathom sets its security posture for a given script based on the following inputs:

***I.1*: Client Policy.** Once users have installed Fathom, they may tune the client policy by customizing a general local security policy that comes with Fathom. This policy specifies the conditions under which Fathom makes API calls available to scripts. For groups of APIs, the policy defines which destinations they may access, whether they require permission to execute, and the form of that permission. Permission forms include user confirmation,

Resource Schema	Description
`http://*.google.com:*`	`fathom.http.*` APIs may contact any *.google.com name on arbitrary ports
`*://{upnp}:*`	any API can contact hosts found via `fathom.upnp.*` calls
`udp://10.1/16:53,5353`	`fathom.udp.*` APIs may contact hosts within 10.1/16 on the DNS and mDNS ports

Table 2: Example resource schemas in Fathom script manifests.

script manifests, and code signing, with multiple forms possible in order to indicate priorities. Users can adjust this policy at any time. Table 1 shows the default policies for the various API groups.

I.2: **User Confirmation.** Fathom can directly ask users whether it should allow some activity at script run time. We do not consider this a particularly appealing mechanism because (*i*) framing questions in a way users will understand is challenging, and (*ii*) a system that overly interferes with a user's workflow may lead users to disable Fathom, or train them to grant permission requests blindly [16]. (Note that Fathom's status icon in the browser's toolbar always indicates to the user when a web page employs Fathom APIs.)

I.3: **Script Manifests.** Each Fathom script includes a *script manifest* that succinctly declares desired resources. Fathom ensures that in-page JavaScript code can only access resources declared in the manifest, subject to the effective client policy. Currently, these resources include Fathom APIs and active measurement targets. APIs and their permissible targets get expressed in tandem in *resource schemas* of the following form:

$$[API\_subset] : //[destinations] : [ports]$$

A resource schema requests access to certain destinations and ports for a subset of the Fathom API. Similar to import statements in other languages, Fathom supports API aggregation, allowing authors to request access to specific functions (e.g., to `fathom.proto.dns.query()`) or to all routines in a family (e.g., `fathom.proto.dns.*`). Destinations come in three forms: (*i*) domains with wildcarding (such as `*.a.com`), (*ii*) IP address ranges (e.g., 192.168/16), and (*iii*) *derived* IP addresses (e.g., `{upnp}`), covering cases where one Fathom API returns an address that the author wants to subsequently pass to another API, but which the script author cannot specify in advance.[3] Table 2 shows examples of Fathom resource schemas.

Fathom enforces resource schemas by initially only making a single API function available: `fathom.init()`. This function receives the script manifest (in JSON form), and subsequently hooks only the requested APIs into the `window.fathom` object. Scripts thus have no way to access APIs they did not request.

I.4: **Server Manifests.** In addition to script manifests, we allow targets to explicitly control permissible incoming traffic via *server manifests*. These manifests reside at a well-known location on web servers, currently `/fathom.json`. Server manifests can whitelist connectivity for scripts originating from specific domains or IP addresses. Users can configure their client policy to allow APIs to proceed without prompting whenever server manifests permit inter-

---

[3]For example, UPnP's discovery phase produces IP addresses to which the script may want to connect via HTTP in order to retrieve a device's service profile. Resource schemas allow precise expression of such dependencies, in contrast to broad notions such as "local network." Fathom tracks hostnames and IP addresses returned by its APIs.

action with the given destination. This scenario forms the default for most APIs initiating connections.

I.5: **Code Signing.** Finally, experimenters can sign Fathom scripts cryptographically to express authorship. Such signatures provide two key benefits. First, signatures establish an identity behind Fathom code, which allows users to reason about scripts in terms of their trust in the author, instead of (or in addition to) making judgments based on the resources the script intends to use (API calls and targets). This trust in authorship likely proves easier for non-technical users to understand. Second, signed code allows users to configure permissions at the granularity of author identities instead of script origin. Furthermore, Fathom can identify scripts abstractly, across origins, and even when users save Fathom-bearing pages to disk.

Examples: With the above inputs Fathom can derive effective policies for each arriving script at run time. For instance, Fathom can let an arriving script proceed without user prompting if (*i*) the client policy (*I.1*) permits execution without prompting for the API resources the script requests (via its script manifest, *I.3*) and (*ii*) the desired destination agrees via a server manifest (*I.4*), given (*iii*) this server manifest actually exists and indeed agrees. In a more simple scenario, if a script bears the signature (*I.5*) of an author whom the user added to their local whitelist indicating full trust in the code, then the script will execute without any prompting.

By design, policy inputs can come into conflict. The client policy resolves such cases. Consider a script that declares (via *I.3*) its need to use `fathom.socket.tcp.*` to open a connection to a given target that has agreed to such communication (via *I.4*). If the user's client policy (*I.1*) requests prompting for permission to use `fathom.socket.tcp.*` then Fathom will prompt the user (via *I.2*) before running the code regardless of the server's consent and any possible signing.

Our hope is that with more extensive experience, a subset of inputs will emerge that proves sufficient, since fewer mechanisms would simplify explication and maintenance of the security framework. Moreover, expanded user experience will allow us to better understand how Fathom's input mechanisms can accommodate the users' potentially limited awareness of the implications of the different permission levels.

Comparison: Fathom's security model borrows from the Java and Flash platforms, with careful consideration of current browsers' same-origin policies. Like Java, we employ code signing and provide user consent as a means to enable functionality, but in contrast to Java's *carte blanche* permissions after user consent to signed code (§ 5.1), Fathom retains precise API-level control and allows neither execution of arbitrary binaries nor file system access. Like Flash, we employ server manifests to establish an explicit consent signal for the Fathom core that permits cross-site connections [33]. Script manifests exist in several technologies, including Java, Flash, and Android apps, as well as Firefox and Chrome extensions. However, our notion of resource schemas renders resource control more explicit than any of these technologies.

5 Implementation

Web browsers are complex systems, hence the design of a browser-based measurement platform requires careful thought. We first discuss the implementation choices within the browser, and then we describe our prototype of Fathom, which we implement as a Firefox extension.

5.1 Implementation space

To maximize ease of adoption, we would like Fathom to depend on the browser's architecture as little as possible while still imple-

Technology	Rich APIs	Accuracy	Portability Browser	OS
Standard JS	✗	Low	✓	✓
Extension (JS)	✓	Good	✗	✓
Plugin (runtime)	✓	High	✗	✗
Plugin (custom)	✓	High	✗	✗
Browser	✓	High	N/A	✗

Table 3: Design space for a browser-based platform. Note: the accuracy of JavaScript-based extensions depends upon the underlying extension API.

menting the design discussed above. In the following, we discuss several points along this design space. Table 3 summarizes the capabilities of each approach.

Standard JavaScript: While it is possible to implement some types of measurements in JavaScript (see discussion in § 7.1), the lack of low-level socket interfaces and access to host properties, leaves this approach suboptimal. Additionally, measurement accuracy would require careful implementation as timer accuracy depends heavily on browser architecture and load.

Browser extensions: Extensions enhance the functionality of the web browser and allow implementation of measurement tasks in portable JavaScript, platform-dependent C/C++, or both. The API richness available to JavaScript extensions varies considerably across browsers. Those extensions execute with browser privileges and have access to most system resources, including the network and file systems, and remain portable across different operating systems for a particular browser. Since most modern browsers compile JavaScript to native code using JIT compilers, extensions do not impose any significant performance overhead. JavaScript extensions present good candidate vehicles for realizing a measurement platform, assuming sufficient API richness and accuracy. C/C++ extensions increase measurement capability at the expense of portability.

Runtime plugins: In this category, Java, Flash, and Silverlight technologies allow developers to implement experiments and easily distribute them to users who have the required runtime plugin installed. Indeed, Java is a popular choice for measurement experiments due to its rich APIs and performance, but it comes with crucial shortcomings that largely also apply to the Flash platform. First, Java's sandboxing keeps execution effectively fully separate from the containing web page. By contrast, measurement code that executes within a web page can directly interact with current web applications. Second, the runtimes' security policies tend to prohibit activities that could aid network measurements (such as `traceroute` or `ping` invocations). Java's code-signing feature provides a way to work around this limitation: after the user accepts an initial dialog presenting the code author's identity, even self-signed applets may access arbitrary runtime APIs and thus behave like host-native applications *with full user privileges*, including permission to create and delete files or contact arbitrary network locations. While no doubt useful in principle, this *carte blanche* ability far exceeds the permissions required to implement measurement and troubleshooting code. Third, the availability of the runtime presents a limiting factor, as virtually no modern web sites use Java applets.[4] Fourth, even when installed, code portability remains uncertain. For example, JavaScript interaction from plugin code remains highly platform-dependent, and the Android operating system supports only an API-limited dialect of Java.

---

[4] OSX and Windows no longer include a Java runtime by default, while several recent critical security incidents relating to Java [17, 26, 35] further erode users' willingness to install the platform.

Custom plugins: We could develop our own native measurement plugin in C/C++. While enabling great performance and host access, portable interaction with the browser remains uncertain. For example, Microsoft recently dropped support for the cross-browser NPAPI [53] in its browsers. Sandboxing technologies like NaCl [55] and Xax [13] allow the execution of native applications within the browser without compromising on performance, but their sandboxed environment respects the browser's security policies and disables application code from performing TCP/IP or UDP networking, or even local file I/O.

Browser changes: One could build the entire platform into an open-source browser, which would provide ideal feature support. This approach would present a substantial development effort and would likely find little immediate support from browser vendors. While not immediately feasible, if a general purpose measurement platform proves useful—especially to web site operators—the functionality could be moved into browsers in the future.

After considering the major browsers and the design space sketched above, we chose to implement our prototype of Fathom as a JavaScript-only extension for the Firefox browser. Several aspects informed this choice: (*i*) Firefox enjoys great popularity and exists on a wide range of different platforms; (*ii*) its XPCOM interface provides a rich API to extensions; (*iii*) as we describe further below, this API allows us to perform low-level socket I/O directly from JavaScript. The latter currently sets Firefox apart from other browsers, for which we would need to implement at least parts of Fathom in native code. The extension realizes the design details outlined in § 4 and currently consists of 3,200 lines of JavaScript.

5.2 Firefox's architecture

Before describing Fathom's architecture in detail, we briefly explain the major components of the Firefox browser relevant to Fathom's design. Firefox's DOM subsystem creates an abstract representation of the visited web pages, which the rendering engine uses for layout and page imaging. A single, shared JavaScript engine parses and interprets both web page and extension code, the browser's *UI thread*. While the browser restricts web page code to stringent security policies, the extension code runs with browser-level privileges. The network engine implements application-layer network protocols such as HTTP and FTP. To do this, it employs the Netscape Portable Runtime (NSPR), Mozilla's platform-neutral API for low-level network I/O and *libc*-like functions. The XP-COM subsystem implements a number of core architecture facilities such as threads and memory management. It also exposes these features to the JavaScript runtime. Furthermore, it includes *js-ctypes* [40], a JavaScript wrapper to the *ctypes* library, allowing extensions implemented in JavaScript to call functions in shared C libraries.

5.3 Architectural challenges

Firefox's architecture poses a number of challenges for extension development, particularly given the design goals for a network monitoring platform as we outlined in § 3.

First, without the *js-ctypes* wrapper, we could not perform socket-level I/O from JavaScript and would need to resort to native code. Even given *js-ctypes*, implementation of some socket APIs requires handling platform dependent data types (such as `union`-based network address structures) which creates a challenge for achieving a truly OS-neutral Fathom. We overcame this problem by providing sufficient padding in the relevant JavaScript datatypes to accommodate varying `union` sizes on different platforms.

Second, despite low-level socket access, ICMP responses and raw sockets remain unavailable. Thus, Fathom cannot provide access to packet-level traces, which would be useful for detailed diag-
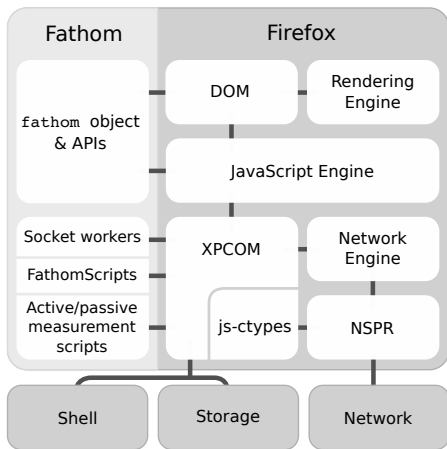
Figure 2: Fathom's architecture in Firefox. Fathom augments the browser's XPCOM subsystem with measurement capabilities and extends each web page's DOM with a `fathom` object and associated measurement APIs. Worker threads enable asynchronous API implementation.

Figure 3: Asynchronous callflow in Fathom between the main thread and a socket worker, for a `recv()` invocation on a UDP socket.

nosis. Fathom overcomes the problem by leveraging access to the browser's command execution API to directly invoke select services on the host, like *ping* or *traceroute* and sidestep this architectural challenge.

Third, JavaScript executing as part of a web page or browser extensions accesses the objects on the web page from the browser's UI thread. Call blocking or long-running functions while executing in this thread would render the entire browser's UI unresponsive.

Fourth, while Firefox extensions can use a threading mechanism called Chrome Workers[5] [39], these threads do not follow the standard OS model of a shared address space with synchronization primitives. Instead, a message-passing system similar to Web Workers exclusively manages communication with worker threads, imposing additional latency.[6]

Fifth, to conserve system resources, recent Firefox versions clamp callback timer accuracy for in-page JavaScript to low precision when the user is viewing a page in a different tab. This clamping can add further function call latency.

5.4 Fathom's architecture

Figure 2 shows the schematic architecture of Fathom running in Firefox. Once installed, the extension creates a component in the XPCOM subsystem that performs two main tasks. First, it installs measurement scripts that handle data collection for passive in-browser measurement. Second, it provides measurement and network I/O APIs to in-page JavaScript. We next discuss the main implementation decisions of Fathom's prototype.

Asynchronous API callflow: As mentioned in § 5.3, in-page and extension JavaScript code as well as the application's UI thread share a single thread of execution. Since Fathom likewise consists of JavaScript code, its focus on network I/O requires careful API design on our part to avoid the possibility of rendering the entire application unresponsive. In order to invoke potentially blocking
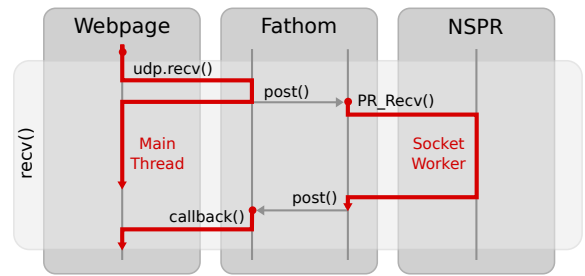
I/O calls without delaying the rest of the browser and without suffering delays due to browser load, Fathom performs *all* such calls in independently executing Chrome Worker threads and uses asynchronously invoked user-provided callbacks when I/O operations complete. Fathom allocates these worker threads on per-socket granularity as *socket workers*. Firefox's event management on such threads thus ensures that I/O operations queue up and get processed as I/O progresses. Figure 3 illustrates the callflow.

Timer throttling: The throttling of frequently executing timer callbacks in background tabs, as mentioned in § 5.3, could affect measurements implemented in inactive in-page JavaScript code. For example, a measurement in a background tab that invokes the `fathom.socket.udp.receive()` API call in a callback invoked periodically via JavaScript's `setInterval()` mechanism, Firefox would clamp the timeout at 1000ms, perhaps significantly slowing down and affecting the measurement. As a workaround, Fathom users can package an entire measurement into a single function (with nested state and functions) and provide the *source* of that function—a *FathomScript*—to Fathom (JavaScript's `toSource()` feature makes this easy). Fathom then launches a low-privileged Web Worker, prepares it such that it offers the Fathom API, and proceeds to execute the function in the Worker. This guarantees independence from in-page timer clamping. Script authors need only be aware of this feature in special cases requiring the use of low-latency I/O inside Fathom (see § 7.1).

Maintaining baseline measurements: For passive measurements, the current implementation of Fathom samples browser and system information every 5 seconds and makes the last 2,000 samples available to all web pages. Each sample takes approximately 450 bytes. While the Fathom implementation stores a sliding window with the last 20 user sessions per domain for baseline metric measurements, all the parameters of the baseline metrics are configurable. We plan to evaluate different techniques for keeping baseline measurements in our future work.

Security model: Our present implementation supports all of the approaches outlined in § 4.3 other than code-signing, which we anticipate adding in the near future. While Firefox already supports signing JavaScript code [38], we have opted not to employ the mechanism directly in Fathom, due to technical shortcomings.[7] We do, however, follow Firefox's approach of signing the JavaScript code of an entire web page, including scripts it pulls in additionally, which provides protection against runtime page modifications.

Controlling resource consumption: Our design incorporates

---

[5]"Chrome" here refers to a privilege level within the Firefox browser, not to the Google Chrome web browser.

[6]While XPCOM exports an interface to manage OS-level threads, we opted for Chrome Workers as they provide a natural, Web Worker-like asynchronous interface and also give thread safety guarantees.

[7]Firefox requires bundling the required files into a JAR file, requires a `jar:` protocol handler for the download, insists on specific `Content-Type` header MIME values, and uses a rather simplistic dialog to convey signer identity to the user.

tracking Fathom's resource usage—e.g., number of open connections or amount of data transmitted and received in the last time unit. Currently we do not restrict resource usage, which is consistent with other runtime plugins such as Flash and Java. As we accrue experience with Fathom we may decide exposing resource usage policy to users and/or developers as it becomes necessary. In the current implementation, if a user judges that a given experiment is overloading her system, she can simply close the web page. Given how simple it is for users to stop measurement pages, experimenters have a strong incentive to use system's resources conscientiously.

6 Performance Evaluation

We evaluate Fathom's performance using two metrics—overhead on web browsing and timing accuracy. We conduct all experiments with Firefox v7.0.1 on a 2.20GHz Intel Core2 Duo machine with 2GB RAM, running Ubuntu 10.04, and connected to a 100 Mbps local network.

6.1 Overhead

We evaluate Fathom's page load overhead for eight popular web sites (CNN, Craigslist, ESPN, Google Maps, New York Times, Slashdot, Yahoo! and YouTube) with diverse JavaScript complexity and number of HTTP requests. For each site, we compare the time interval between initiating a page load and the firing of the `Load` event in a Fathom-enhanced browser against a normal browser. The `Load` event fires at the end of the page loading process when all page components have rendered. To obtain the ground truth, we enhance the browser with a lightweight extension that captures only the page load times. We also compare Fathom's overhead with that of Firebug [36], a popular browser extension to debug web page performance. Firebug performs extensive diagnostic measurements, which increase the page load time. We use Firebug as a reference and not as a direct comparison.

We perform 50 page loads for each benchmark and record the timestamp values for the `Load` event and the clock-time spent within Fathom's code for collecting the metrics. To measure the worst case performance overhead Fathom imposes, we conduct all experiments on a browser with a warm cache. Table 4 shows the average absolute load times and Fathom's relative overhead for the benchmarks. The count of HTTP requests, scripts, and images indicates the range between the minimum and the maximum number of requests of each type until the page finishes loading. Requests that start after the page has loaded are not included in these numbers, as Fathom's activity for such requests will not affect the load time.

Overall, Fathom imposes 1–3% overhead for all the benchmarks. We note that as the number of HTTP requests increase, the average execution time within Fathom also increases. We expect this behavior, because Fathom observes all HTTP activity to collect passive measurements. We also observe that with the increase of HTTP requests, Fathom does not always incur a corresponding increase in the absolute overhead. This is because Fathom's overhead is not additive. Since we implement Fathom in JavaScript, its execution will only block the single-threaded JavaScript interpreter, while the rest of the browser can continue loading the page. Thus, Fathom's overhead gets masked in part by network latency and the browser's parallelism during the document loading process.

6.2 Accuracy

Network measurement platforms require accurate timestamps and timers. Fathom leverages the browser's JavaScript API to record timestamps and the XPCOM API to implement a timer. As with all

	No browsing		Passive browsing		Active browsing	
	Avg.	Stdev.	Avg.	Stdev.	Avg.	Stdev.
Send (ms)	0.52	0.29	0.54	0.28	0.52	0.39
Recv (ms)	0.41	0.33	0.50	0.75	0.54	0.89

Table 5: Difference in timestamps from Fathom and `tcpdump` for sent and received packets under different browsing conditions.

measurement, competing activity (browser, host and network) can affect accuracy of timestamps and firing of timers.

Timestamp accuracy: We measure the accuracy of Fathom's timestamps by sending probe packets between two machines on the same local network and recording timestamps for those packets in Fathom and via packet capture with `tcpdump` on the same machine. Since the timestamps from Fathom and `tcpdump` stem from the same clock, the difference between the timestamps for each packet are due to the time required to traverse the network stack between the interface and the browser.

We test timestamp accuracy under different browsing conditions with a train of 106 probe packets with a payload of 200 bytes each. In the "no browsing" scenario, there is no other browsing activity in progress. For the "passive browsing" scenario, we load four web sites (CNN, Google Maps, NYTimes and YouTube) in separate browser tabs and send the probe train after the sites finish loading. For the "active browsing" scenario, we send the probe train while simultaneously loading the same four sites in other tabs.

Table 5 shows the results of the experiment. The "send" row indicates the difference in timestamps from the time Fathom dispatches the probe to the resulting timestamp from `tcpdump`. Similarly, the "recv" row indicates the delay from the timestamp given by `tcpdump` to its arrival in Fathom. In all the browsing scenarios we find the accuracy of the timestamps in Fathom—which inherit the XPCOM granularity of 1 ms—to be within 1 ms of the `tcpdump` reported event time. Further, the standard deviation for all experiments is also below 1 ms. In the passive and active browsing scenarios, Fathom's receive timestamp accuracy remains lower than in the "no browsing" case, but the accuracy is still less than 1 ms.

We next explore Fathom's timestamp accuracy under heavy cross-traffic on the host. We repeat the experiment for 10 probe trains of 10 packets while performing an *iperf* [21] transfer with 4 TCP connections between two hosts on the same local network. In this experiment, the average difference in the "send" timestamps increases to 729 ms. This test represents a worst-case scenario when users fully utilize their network; the heavy concurrent network traffic from *iperf* completely fills the egress queue, which prevents timely transmission of Fathom-generated probes.

Timer: We measure the accuracy of Fathom's timer by testing how precisely it can dispatch 200-byte probes at 200 ms intervals. We use the timer to send 51 probe trains of 100 probes each and measure the average (μ) and standard deviation (σ) for the observed intervals between consecutive probes. Since the accuracy for XPCOM timers is 1 ms, a σ of more than 1 ms would mean that the timer loses precision. We study timer accuracy in four scenarios: (i) no browsing, (ii) passive browsing, as described above (and for which asynchronous activity due to both ads and scripts can lead to concurrent activity), (iii) active browsing, as described above, and (iv) CPU stress case, where we emulate a competing process in the host that overloads the CPU.

For the no-browsing and stress tests, we find the mean timer interval to be within 1 ms of the expected value of 200 ms. Further, for both cases the 5th percentile was 199 ms and the 95th per-

Benchmark	# HTTP	# Scripts	# Images	Avg. load time (ms)	Avg. Fathom load time (ms)	Avg. Firebug load time (ms)	Avg. execution time within Fathom (ms)	Avg. Fathom overhead(%)	Avg. Firebug overhead (%)
Craigslist	4	2	0	512	517	532	9	1.1	4
YouTube	28–34	2–3	17–23	869	896	957	37	3.1	10
Google Maps	39–40	7–8	26–28	1233	1268	1585	60	2.9	29
Yahoo	49–58	4–5	31–43	1168	1180	1747	42	1.0	50
Slashdot	35–60	2–12	22–30	2381	2414	4083	51	1.4	71
ESPN	85–89	7–8	61–63	1761	1805	2325	96	2.5	32
CNN	89–104	6–9	22–59	1458	1504	2263	174	3.1	55
NY Times	115–125	23–26	77–80	1144	1159	1798	91	1.3	57

Table 4: Average page load overhead of Fathom for popular web pages over 50 runs.

centile was 201 ms. For the passive and active scenarios, we use 10 probe trains of size 10, with probes sent at 200 ms intervals. In the passive-browsing scenario, the average lies within 2 ms of the expected, the 5th percentile was 1 ms, and the 95th percentile was 267 ms. For the active browsing scenario, the average lies within 33 ms of the expected, the 5th percentile was 12 ms, and the 95th percentile was 669 ms. This large drop in accuracy compared to the no-browsing and stress experiments occurs because multiple tabs in the same browser window share the same browser runtime.

To sum up, Fathom often achieves 1 ms timestamp accuracy, but the accuracy of timestamping sent packets degrades under both heavy network traffic and concurrent tabs within the browser. The degradation in timing accuracy due to competing activity is a well-known difficulty when measurements are running in shared environments like end-systems. For example, it is well-known that measurements running in PlanetLab will suffer due to concurrent activity [48]. Measurement scripts that use Fathom should be aware of these issues and perform extra tests to verify the accuracy of their experiments. For example, scripts can invoke Fathom APIs to request CPU load and network cross-traffic on the host to test existing operating conditions.

7 Case Studies

In this section, we demonstrate Fathom's utility by implementing three example usage cases that together exercise different aspects of its API: providing a JavaScript version of ICSI's Netalyzr testsuite, debugging web access failures, and enabling Google Maps to diagnose poor web page performance. Our goal is to illustrate how Fathom can support disparate measurement needs.

7.1 Netalyzr

We test Fathom's suitability as a platform for conducting a wide range of active measurements by reproducing many of the tests present in our Netalyzr testsuite [28]. Netalyzr relies on a Java applet for its client-side tests, so supporting this range of measurements can serve as a litmus test for Fathom's flexibility. We developed a Fathom-powered JavaScript version of the suite,[8] focusing in particular on tests not feasible in standard JavaScript.

API requirements: In the following, we consider Netalyzr's full set of test categories and discuss Fathom's significance in the tests.

Address-based tests: This category primarily investigates whether the client resides behind a NAT, and if so, how the NAT renumbers the client-side address and port numbers. It requires the equivalent of `getsockname()`, i.e., access to the local address and port of a socket. Standard JavaScript does not provide these, but Fathom's TCP APIs do.

UDP and TCP port reachability: Tests in this category measures the extent to which the client can communicate freely on approximately 30 ports used by standard protocols. For TCP, standard JavaScript APIs can approximate these tests somewhat clumsily,

---

[8]`http://netalyzr.fathom.icsi.berkeley.edu`

while for UDP no mechanisms exist. Fathom provides raw TCP and UDP access, making these tests easy.

Access link properties: Netalyzr leverages Java's raw UDP access to implement basic latency testing, upstream/downstream bandwidth measurement, and in-path buffer size measurements. Standard JavaScript APIs provide no equivalent functionality, but Fathom's UDP access allowed us to realize the tests.

HTTP functionality & correctness: These tests determine the presence of HTTP proxies and, if found, the use of caching and caching correctness. The JavaScript XMLHttpRequest API is insufficient to implement Netalyzr's proxy tests as it provides inadequate control over raw message headers and browser caching. Fathom's TCP APIs allow us to realize these tests.

DNS functionality & correctness: This category broadly covers DNS feature support tests, lookup performance measurements, lookup result correctness, and tests for known security problems. While standard JavaScript APIs can use the success status of HTTP requests as an implicit signal of success for the underlying DNS lookup, one cannot obtain the actual IP address resulting from the request, which virtually all tests in this category require. Fathom's DNS API allows us to conduct these tests.

IPv6 support: This category includes reachability tests to IPv6-only servers as well as performance comparisons to servers reachable over IPv6 as well as IPv4. As the address family effectively remains abstracted, standard JavaScript can handle these tests via URL requests from the servers in question.

API usability: The Java-to-JavaScript porting effort also allow us to compare programming paradigms. JavaScript's basic single-threaded execution model with callback-driven asynchronicity differs substantially from Java's lightweight threading model. Accordingly, restructuring the client's overall execution flow required significant reworking of the basic design, while porting the tests themselves felt quite straightforward. Netalyzr's tests execute in individual threads using I/O that may block, with a main thread monitoring test durations. In writing our JavaScript version, we restructured each test into a larger set of individual functions chained together via callbacks invoked by Fathom upon I/O progress. The main test sequence likewise leverages test completion callbacks. We implemented the Fathom-driven Netalyzr subset in two weeks, the majority of which went into replicating the overall test execution, reporting harness, and GUI. Each test took on the order of an hour to port.

Fathom bandwidth tests: Netalyzr's testsuite includes measurements of the available send and receive bandwidths. We implement two versions of the bandwidth test:

- **In-Fathom.** To estimate the maximum possible send/receive rate that Fathom can achieve, we invoke NSPR's UDP `send`/`recv` APIs within a tight loop to saturate the available network bandwidth. Since executing a tight loop increases CPU load, we repeat the experiment and invoke the NSPR `send`

Send / Receive	Payload (bytes)	Iperf (Mbps)	In-Fathom (Mbps)		Web page (Mbps)	
			Loop	Callback	Loop	Callback
Send	**0**	13.60	6.64	2.42	1.66	0.75
	1024	93.90	92.35	81.41	49.48	24.51
Recv	**0**	4.89	4.55	0.79	0.84	0.02
	1024	93.70	73.36	52.85	22.63	1.69

Table 6: Fathom's send/receive data rates for different payloads.

API within a `setTimeout` with delay 0, which enables other events in the event queue to get processed.

- **Web page.** To get an estimate of the send/receive traffic that JavaScript on a web page can achieve, we perform the same tests but invoking Fathom's socket `send`/`recv` APIs, both within a loop and using a timer.

We perform all tests between a laptop and a desktop connected over a 100 Mbps Ethernet and compare them with the speed achieved by *iperf*. We determine the maximum available send bandwidth using a Fathom client and an *iperf* server. We determine the maximum available receive bandwidth using a Fathom-based server and an *iperf* client. Table 6 shows the results. We observe that under the In-Fathom category, using the NSPR APIs we get much higher traffic rates than the web pages using Fathom's APIs. We expect this, since NSPR APIs are synchronous, while due to the asynchronous nature of Fathom APIs every additional callback in the call sequence affects the bandwidth rates.

We also observe that send rates increase as the payloads increase and match *iperf* send rates at high payload size. But the maximum achievable receive rate did not exceed 73.4 Mbps. Although we could not experimentally verify the cause of the imbalance in the send and receive rates, the discrepancy could arise due to the js-ctypes library used for Fathom's socket APIs. We suspect js-ctypes, which does not benefit from JavaScript's JIT optimizations [18, 6], may pose a significant performance bottleneck.

A natural way to improve the send/receive bandwidths is to parallelize the tests. We tried using HTML5 Web Workers, which spawn OS-level threads to concurrently perform the bandwidth tests. We observed that the reported bandwidth did not increase when running the test in a tight loop. We attribute this to a limitation of the Workers, which cannot access the page's JavaScript namespace, including the `window` object. Workers, thus, cannot access Fathom APIs defined on the `window.fathom` object. All accesses to Fathom APIs must require another level of indirection using the asynchronous message passing `postMessage` API between the Worker thread and the main UI thread (which has access to the `window.fathom` object). This indirection limits the achieved bandwidth.

To enable better bandwidth tests from a web page, script authors should therefore implement bandwidth tests via FathomScripts (recall § 5.4), as they obviate the message passing overhead.

Summary: Our implementation of key Netalyzr tests that prove difficult or infeasible to realize in standard JavaScript (local network configuration, UPnP probing, DNS features, and particularly UDP-based latency and bandwidth tests) confirm Fathom's versatility. For tests leveraging low-level APIs, particularly `fathom.socket.*`, the coding effort compares to that required in Java, while tests that could leverage Fathom's APIs (e.g. `fathom.proto.upnp.*`) instead of needing full protocol implementations required a fraction of the code.

7.2 Web access failure debugging

While troubleshooting suites such as Netalyzr work well when the user has access to sufficient connectivity to download and run the testsuite, such an approach fails in the presence of more fundamental connectivity problems. Indeed, a frequent request we receive from Netalyzr users is to expand the connectivity tests in that direction, particularly in Netalyzr's command-line interface. For such scenarios, we developed a "Debug my connection" measurement application. We include this script as a built-in part of the Fathom distribution (which provides an immediate incentive for users to install Fathom), but emphasize that we implement it solely in JavaScript using the existing Fathom API. The script is currently 400 LOC.

Implementation: We structure the connectivity debugger by starting at the host's link-level network interface configuration and progressing upward through the protocol stack, identifying and reporting problems at each layer. The test begins by obtaining basic system usage statistics from Fathom's usage baseline, including memory usage and process load. Using `fathom.system.getActiveIntefaces()`, we detect whether the system actually connects to any network. If so, we report recent link quality and throughput counters for the active interfaces, as reported by `fathom.system.getNetworkUsage()`, and, again using the baseline counters, report on cross traffic originating from outside of the browser. We then move up to the network layer and test the availability of a default route in the system's routing table, as well as IP-level connectivity (via `fathom.system.doPing()`) to the gateway and DNS resolver(s) (via `fathom.system.getResolvers()`), and relevant browser-level configuration settings such as proxies (via `fathom.utils.browser.getProxyConfig()`). Moving up to the transport level, we attempt retrieval of an image from `www.google.com`, and the test completes.

At each stage, we report on the outcome of the tests involved. In the current version, users click on a "Debug my connectivity" button provided in the browser toolbar in order to initiate diagnosis, but we envision hooking the connectivity debugger into relevant parts of the browser in order to automatically provide contextual information to the user. For example, upon failing DNS lookups, we can automatically test the DNS configuration, and for unavailable servers we can test whether the user experiences a full outage or one just affecting the destination site.

Collaborative debugging: We can further troubleshoot possible connectivity problems in the local network by leveraging Fathom's rendezvous and server-side abilities. In particular, we can leverage Fathom's UPnP and zeroconf capabilities (via `fathom.proto.{mdns,upnp}.*`) as follows. First, using UPnP we can try to locate UPnP-enabled gateway devices. If available, such devices can report uplink type, upstream/downstream throughput counters, and other information [51]. Second, using UPnP and Zeroconf we can locate *other* Fathom-enabled devices. Since Fathom supports listening sockets, we can then engage in both local and remote latency and throughput tests. Correlating the resulting numbers allows us to pinpoint packet loss problems.

To test this approach, we have implemented a Fathom-powered web application in 320 lines of code that lets two Fathom instances detect each other using UPnP and initiate probes to measure unidirectional loss and latency between each pair of Fathom-enabled nodes. The script then compares these values with historic values of loss rates and delays for the local networks, obtained from Fathom's passive data collection. Values greater than expected cause the application to report poor local network performance. Crucially, in order to conduct this testing the user neither needs to
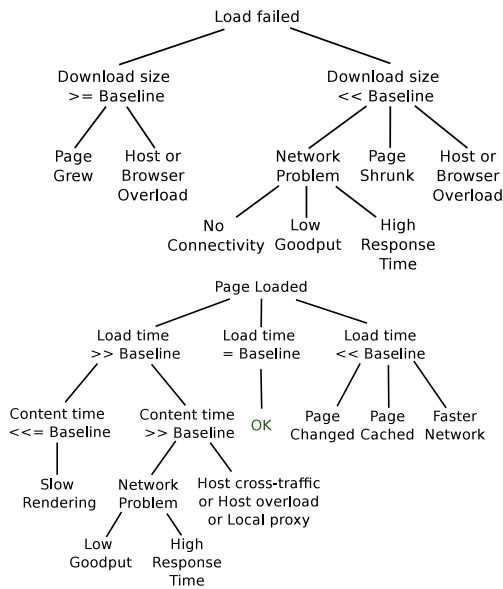
Figure 4: Diagnosis logic when the page load failed (top) vs. when it succeeded (bottom).

7.3 Web services debugging: Google Maps

Fathom substantially expands the ability of web site developers to understand the context and impediments that affect the way their pages render in the client's browser. Fathom's API enables a range of potential measurements of interest in this context, such as pinpointing of loss in the home or backbone [11], system and I/O load on the client's device, and network connectivity configurations that hamper performance (e.g., via slow HTTP proxies). In the following, we demonstrate Fathom's utility in troubleshooting client-side issues for web services by playing the role of the developer for Google Maps. We make two modifications to the Google Maps web page code. First, we embed a diagnostic script that uses Fathom APIs to obtain results for passive and active measurements. Second, we provide a debug button on the Google Maps page, which enables a user to initiate execution of the debug script. In the special case when the button itself does not render (because page loading failed), users can employ Fathom's connectivity debugger as explained in the previous section.

Implementation: The debug script first verifies if the `Load` event for the Google Maps web page fired, confirming full page load. Figure 4 illustrates the decision logic Fathom follows for failures (top) and successes (bottom) of page load attempts. If the *load time*—i.e., the time elapsed between the first HTTP request and when the `Load` event fires—resembles[9] the corresponding baseline value, then the algorithm reports a normal load and terminates.

If the load time turns out far smaller than the baseline value, the baseline metric does not apply to the current scenario. Possible causes include changing content size, page caching, and improving network conditions (such as when the user moved to a faster net-

---

[9]We say that a value is similar to the corresponding baseline value if it lies within $\pm 2\,\sigma$ of the baseline value.

work). Fathom tracks all HTTP-level metrics for the last 20 user sessions. Using DOM `localStorage`, it makes the results available to all scripts of the same origin. We then average these values to generate baseline values for Google Maps. As a result, the debug script can identify the problem by checking if the amount of downloaded content for the web page reduced substantially, if the data rate for individual resources in the web page improved, and also if the browser fetched a web page resource entirely from the cache.

If the load time far exceeds the baseline, it could indicate a network impediment or a host problem. To distinguish between the two cases, the debug script examines the *content time*, i.e., the time elapsed to download the content of the web page. A content time less than or equal to the baseline indicates that a problem with the host machine or the browser slowed down the rendering. Such a condition could also arise if the user explicitly stops the loading of the web page and then invokes the debug script. Our debug script uses `fathom.system.*` APIs to determine resource utilization. If content time proves larger than the baseline, the problem most likely resides in the network. We debug slow networks as discussed in the previous section.

Emulating Google Maps: Since we did not have access to Google Maps servers, we modified the site's JavaScript code at runtime. When visiting the Google Maps page, we inject the debug script and the Fathom debug button using Greasemonkey [30]. Our debug script uses Fathom APIs to implement all diagnostic cases shown in Figure 4.

We implemented Fathom's Google Maps debug script in 640 lines. It imposes no observable runtime overheads. The bulk of the script implements parsing routines to interpret responses from `fathom.system.*` as well as statistical utilities to compute average and standard deviation for metric values. Fathom's system, network, and browser-level APIs directly provide all values except for cross-traffic, which we measure using a combination of APIs: cross-traffic at the host is the fraction of HTTP traffic observed by the browser over the TCP-level traffic approximated from the link-level network measurements. In order to estimate cross-traffic given only transport level information from the browser, Fathom must also account for the effect of framing. To do so, Fathom assumes all link-level traffic to be TCP and subtracts the per-packet overhead from the traffic counters (in bytes) to get approximate transport level TCP traffic. It then compares this value against the TCP traffic values available from the browser.

We used our debug script and performed 10 trials of loading the Google Maps home page with packet loss of 10, 20, 30 and 50 percent and packet delays of 0.5, 1.0, 1.5 and 2.0 seconds. We emulate packet loss and delays with *netem*. In each case, before starting the experiments, we loaded the Google Maps home page repeatedly to allow Fathom to calculate the starting baseline metrics using 20 normal sessions.

While the page load time increased in the presence of packet loss rates up to 20%, the content download time remained within two standard deviations. Higher packet loss rates severely affected the goodput and the page took much longer to load. We observed that for all packet delays (starting from 0.5 seconds), the debug script indicated load times larger than two standard deviations and the content download time also increased. In all cases, the debug script identified a change in the network operating conditions. As the number of completed connection trials increased, more samples fell into the range of two standard deviations and so Fathom could include the HTTP metric values from the newer sessions. The debug script thus adjusted the baseline correspondingly, with the result that some of the later trials reported normal operating conditions.

8 Discussion

Portability: While Fathom currently works only in Firefox, we envision broadening support to other browsers and environments. The Android version of Firefox shares the desktop version's architecture and thus required only minor Fathom compatibility work.[10] In Chrome [3], the implementation would require a combination of content scripts for hooking the API into `window`, an extension core for mediation between web page and OS,[11] and a native binary for all remaining functionality, including invocation of commands on the host. Fathom's callback-driven API would map well to Chrome's architecture. Internet Explorer and Safari prove tricky; the former will require the development of a browser helper object, while the latter's limited extension JavaScript API requires falling back to a plugin component.

Incentives for adoption: Fathom provides strong incentives for end-users, researchers and website operators. For end-users, Fathom's built-in troubleshooting capabilities provide an immediate incentive to install the platform, as a means to help debug connectivity problems that users would otherwise need to solve manually. Our experience with Netalyzr shows that there is in fact a desire for such debugging tools among a set of users who will return to the Netalyzr site to run its set of diagnostics, and who see fit to leave feedback [27]. As Fathom can support troubleshooting tasks both in the presence of fundamental connectivity problems (which preclude the use of website-driven services such Netalyzr) as well as with the ease of visiting a webpage when connectivity permits, we believe this incentive holds at least as strongly for Fathom as it does for Netalyzr. For researchers, Fathom's appeal lies in the fact that it does not require manual installation of code specific to their experiments; once the user has installed Fathom, *any* Fathom-driven experiment will work with the convenience of visiting a web page, lowering barrier-to-entry for both users and experimenters. For site operators, similar incentives apply: once users have installed Fathom, both they and the site operators benefit from increased understanding of performance impediments.

Support for long-running measurements: Quite often experimenters may need to execute long running measurement tasks, such as when studying the evolution of network topology [8, 47] or the dynamics of network performance [5]. Fathom's browser-based nature prompts the question of how long users normally view a particular page before moving on, closing its tab, or closing the browser's window altogether. One dataset collected by the Mozilla TestPilot project and recording browsing behavior of 2,000 Firefox users [41] provides limited insight into this aspect: in general web surfing, 20% of browser windows remain open for at least an hour, 7% for more than a day.[12] With the aid of gentle reminders to users of experiments still actively running in a tab or window, Fathom should thus be able to provide a reasonable level of longer-term data collection. Fathom's current implementation does support execution of such long running tasks if the measurement script is invoked within a *FathomScript* (recall in § 5.4). This mechanism would allow independence from user interaction with the web page (and thus immunize the experiment from potentially reduced scheduling priorities or other service degradations imposed by the browser when a tab does not have focus), but it would still require the end-user to keep the measurement tab open. This is

because Fathom controls resource consumption by deallocating all requested resources when the corresponding browser tab/window is closed. Also, Fathom currently does not save any measurement state in the event of browser or system shutdown. We anticipate introducing a special API for experimenters to launch experiments that can run in the background to avoid tying the longevity of a measurement task to a browser tab. This approach will provide independence from accidental tab closures but requires more careful thought regarding the resource management of long-running measurement tasks. For example, we must consider (i) what experiment state and resources must be preserved across browser or system reboots, and (ii) the security implications of running a background task and providing it with access to host and web resources.

9 Related Work

We divide related work in three main areas: browser-based approaches to profile page performance, server-based measurement platforms, and efforts that piggyback measurements with applications already running on end systems.

Browser-based page performance profiling. A number of browser-based approaches help page developers benchmark and debug web page performance. The Firebug [36] extension and the dynaTrace [14] application break down the time to load pages, but these measurements are only accessible from the browser, not the web page itself. As a result, web page developers cannot remotely execute tests from real users' browsers. The Boomerang JavaScript library [54] overcomes this obstacle and, like Fathom, gives web pages a vantage point directly from the end-system. Janc et al. combined JavaScript and Flash to measure throughput and jitter in HTTP downloads [22, 23]. Neither effort strives to provide a generic API. While the forthcoming NavigationTiming API [52] standardizes latency measurements by providing timestamps for fixed events during a page's loading cycle, it does so only for the page itself, not embedded components. Fathom furthermore adds the ability to contrast passive timing collection with active measurements. Rivet [34], another recent effort, leverages reflection and dynamic characteristics of JavaScript to implement a remote debugger for web applications. As a regular JavaScript library, Rivet does not fundamentally add functionality to JavaScript in web pages. Unlike Fathom, Rivet therefore cannot enable additional passive or active measurement tasks.

Measurement platforms. Researchers often use PlanetLab [9] as a measurement platform. More recently, Google launched M-Lab [32] as another measurement platform. Both PlanetLab and M-Lab are based on well-connected servers, and not on end systems as is Fathom. Scriptroute [49] added the capability for any Internet user to run measurements using scripts in a sandboxed, resource-limited environment in PlanetLab. We share Scriptroute's goal of building a community platform for network measurements. However, our focus on taking measurements directly from end systems brings new challenges in particular: portability across operating systems and small performance overhead.

Piggybacking measurements with applications. Dasu [5] is a Vuze plugin that analyzes BitTorrent traffic to characterize ISP performance. Because Dasu piggybacks on BitTorrent, it has a large user base. BitTorrent's frequent use for sharing pirated content, however, can make its use problematic from a policy perspective. We instead opt for deploying measurements on the web browser, which is even more ubiquitous than BitTorrent. The Measurement Manager Protocol (MGRP) [43] is an in-kernel service that runs on end systems to reduce overhead for active measurement tools. MGRP allows measurement tools to send probe trains where appli-

---

[10]Fathom for the Android platform is available on `http://fathom.icsi.berkeley.edu`.

[11]This could leverage Chrome's upcoming support for TCP and UDP sockets in extensions, see `http://developer.chrome.com/trunk/apps/app_network.html`.

[12]The dataset's collection was limited to one week and thus forms a lower bound for longer-running window instances.

cation packets piggyback on measurement probes. Neither MGPR nor Dasu provide a programmable measurement platform.

10   Conclusion

In this paper, we describe *Fathom*, a browser extension that uses web browser APIs to build a measurement platform. Fathom provides a programmable interface for writing and launching measurements from the convenience of the web browser. We have implemented a Fathom prototype for the Firefox web browser. Our evaluation shows that it can achieve timestamp accuracies of 1 ms, and generally has runtime overheads of $< 3.2\%$ for popular websites. We also demonstrate its utility in three case studies: providing a JavaScript version of the *Netalyzr* network characterization tool, debugging of web access failures, and enabling web sites to diagnose performance problems of their clients.

11   References

[1] S. Agarwal, P. Mohan, N. Liogkas, and V. Padmanabhan. Webprofiler: cooperative diagnosis of web failures. In *Proc. COMSNETS*, 2010.

[2] G. Aggrawal, E. Bursztein, C. Jackson, and D. Boneh. An analysis of private browsing modes in modern browsers. In *Proc. USENIX Security*, 2010.

[3] A. Barth, A. Felt, P. Saxena, and A. Boodman. Protecting browsers from extension vulnerabilities. In *Proc. NDSS*, 2010.

[4] S. Bauer, D. Clark, and W. Lehr. Understanding broadband speed measurements. *MITAS project white paper*, 2010.

[5] Z. S. Bischof, J. S. Otto, M. A. Sánchez, J. P. Rula, D. R. Choffnes, and F. E. Bustamante. Crowdsourcing ISP characterization to the network edge. In *Proc. ACM SIGCOMM Workshop on Measurements Up the Stack*, 2011.

[6] Bugzilla. https://bugzilla.mozilla.org/show_bug.cgi?id=687306.

[7] M. Butkiewicz, H. V. Madhyastha, and V. Sekar. Understanding website complexity: Measurements, metrics, and implications. In *Proc. IMC*, 2011.

[8] K. Chen, D. R. Choffnes, R. Potharaju, Y. Chen, F. E. Bustamante, D. Pei, and Y. Zhao. Where the sidewalk ends: Extending the Internet AS graph using traceroutes from P2P users. In *Proc. CoNEXT*, 2009.

[9] B. Chun, D. Culler, T. Roscoe, A. Bavier, L. Peterson, M. Wawrzoniak, and M. Bowman. PlanetLab: An overlay testbed for broad-coverage services. *ACM SIGCOMM Computer Communication Review*, 33(3), July 2003.

[10] H. Cui and E. Biersack. Trouble shooting interactive web sessions in a home environment. In *Proc. ACM SIGCOMM Workshop on Home Networks*, 2011.

[11] L. DiCioccio, R. Teixeira, M. May, and C. Kreibich. Probe and Pray: Using UPnP for Home Network Measurements. In *Passive and Active Measurement Conference (PAM)*, Vienna, Austria, March 2012.

[12] M. Dischinger, M. Marcon, S. Guha, K. P. Gummadi, R. Mahajan, and S. Saroiu. Glasnost: Enabling end users to detect traffic differentiation. In *Proc. USENIX NSDI*, 2010.

[13] J. R. Douceur, J. Elson, J. Howell, and J. R. Lorch. Leveraging legacy code to deploy desktop applications on the web. In *Proc. USENIX OSDI*, 2008.

[14] dynaTrace Software. Diagnose and prevent AJAX performance issues. http://ajax.dynatrace.com/.

[15] P. Eckersley. How unique is your web browser? In *Proc. Privacy Enhancing Technologies Symposium (PETS)*, 2010.

[16] A. P. Felt, S. Egelman, M. Finifter, D. Akhawe, and D. Wagner. How to Ask For Permission. In *Proc. USENIX Workshop on Hot Topics in Security*, 2012.

[17] FireEye. Zero-day season is not over yet. http://blog.fireeye.com/research/2012/08/zero-day-season-is-not-over-yet.html.

[18] A. Giammarchi. An introduction to js-ctypes. http://webreflection.blogspot.com/2011/09/introduction-to-js-ctypes.html.

[19] Google. Google Maps help forum. http://www.google.com/support/forum/p/maps/thread?tid=24f446d4cc24d07a.

[20] Internet2. Network Diagnostic Tool (NDT). http://www.internet2.edu/performance/ndt/.

[21] Iperf. http://iperf.sourceforge.net/.

[22] A. Janc. Nettest. http://code.google.com/p/nettest/.

[23] A. Janc, C. Wills, and M. Claypool. Network performance evaluation in a web browser. In *Proc. IASTED PDCS*, 2009.

[24] S. Kambala. Speed metrics in Google Analytics. http://googlecode.blogspot.com/2011/12/speed-metrics-in-google-analytics.html.

[25] P. Kanuparthy and C. Dovrolis. ShaperProbe: End-to-end detection of ISP traffic shaping using active methods. In *Proc. IMC*, 2011.

[26] A. Kingsley-Hughes. Flashback Malware Worth Up To $10,000 A Day To Criminals. http://www.forbes.com/sites/adriankingsleyhughes/2012/05/01/flashback-malware-worth-up-to-10000-a-day-to-criminals/, May 2012.

[27] C. Kreibich, N. Weaver, G. Maier, B. Nechaev, and V. Paxson. Experiences from Netalyzr with engaging users in end-system measurement. In *Proc. ACM SIGCOMM Workshop on Measurements Up the Stack*, 2011.

[28] C. Kreibich, N. Weaver, B. Nechaev, and V. Paxson. Netalyzr: Illuminating the edge network. In *Proc. IMC*, 2010.

[29] V. Lam, S. Antonatos, P. Akritidis, and K. Anagnostakis. Puppetnets: misusing web browsers as a distributed attack infrastructure. In *Proc. ACM CCS*, 2006.

[30] A. Lieuallen, A. Boodman, and J. Sundström. Greasemonkey. http://www.greasespot.net/.

[31] M. Mathis, J. Heffner, P. O'Neil, and P. Siemsen. Pathdiag: Automated TCP diagnosis. In *Proc. PAM*, 2008.

[32] Measurement Lab. http://www.measurementlab.net/.

[33] D. Meketa. Policy file changes in Flash Player 9 and Flash Player 10. http://www.adobe.com/devnet/flashplayer/articles/fplayer9_security.html.

[34] J. Mickens. Rivet: Browser-agnostic Remote Debugging for Web Applications. In *Proc. USENIX ATC*, pages 30–38, 2012.

[35] Mitre.org. Cve-2012-4681.
http://cve.mitre.org/cgi-
bin/cvename.cgi?name=2012-4681.

[36] Mozilla. Firebug. http://getfirebug.com/.

[37] Mozilla. Perfmeasurement.jsm. https://developer.
mozilla.org/en/JavaScript_code_modules/
PerfMeasurement.jsm.

[38] Mozilla. Signed Scripts in Mozilla.
http://www.mozilla.org/projects/
security/components/signed-scripts.html.

[39] Mozilla Developer Network. ChromeWorker.
https://developer.mozilla.org/en/DOM/
ChromeWorker.

[40] Mozilla Developer Network. js-ctypes.
https://developer.mozilla.org/en/js-
ctypes.

[41] Mozilla TestPilot. Tab Switch Study.
https://testpilot.mozillalabs.com/
testcases/tab-switch-study.

[42] V. N. Padmanabhan, S. Ramabhadran, S. Agarwal, and
J. Padhye. A study of end-to-end web access failures. In
Proc. CoNEXT, 2006.

[43] P. Papageorge, J. McCann, and M. Hicks. Passive aggressive
measurement with MGRP. In *Proc. ACM SIGCOMM*, 2009.

[44] V. Paxson. End-to-end routing behavior in the Internet. *Proc.
ACM SIGCOMM*, 1996.

[45] M. Perry. Torbutton design documentation. https://
www.torproject.org/torbutton/en/design/.

[46] C. Reis, S. Gribble, T. Kohno, and N. Weaver. Detecting
in-flight page changes with web tripwires. In *Proc. USENIX
NSDI*, pages 31–44, 2008.

[47] Y. Shavitt and E. Shir. DIMES: Let the Internet measure
itself. *ACM SIGCOMM Computer Communication Review*,
35(5), 2005.

[48] J. Sommers and P. Barford. An active measurement system
for shared environments. In *Proc. IMC*, 2007.

[49] N. Spring, D. Wetherall, and T. Anderson. Scriptroute: A
public Internet measurement facility. In *Proc. USENIX
Symposium on Internet Technologies and Systems*, 2003.

[50] J. Ullrich. Javascript DDoS Tool Analysis.
http://isc.sans.org/diary/Javascript+
DDoS+Tool+Analysis/12442.

[51] UPnP Forum. Internet Gateway Device (IGD) V 2.0.
http://upnp.org/specs/gw/igd2.

[52] W3C. Navigation timing, editor's draft november 14th, 2011.
https://dvcs.w3.org/hg/webperf/raw-file/
tip/specs/NavigationTiming/Overview.html.

[53] Wikimedia Foundation, Inc. Netscape Plugin Application
Programming Interface.
http://en.wikipedia.org/wiki/NPAPI.

[54] Yahoo! Exceptional Performance Team. This, is boomerang.
http://yahoo.github.com/boomerang/doc/.

[55] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Orm,
S. Okasaka, N. Narula, N. Fullagar, and G. Inc. Native
Client: A sandbox for portable, untrusted x86 native code. In
Proc. IEEE S&P, 2009.