Embassies: Radically Refactoring the Web

Jon Howell, Bryan Parno, John R. Douceur, *Microsoft Research*

Abstract

Web browsers ostensibly provide strong isolation for the client-side components of web applications. Unfortunately, this isolation is weak in practice; as browsers add increasingly rich APIs to please developers, these complex interfaces bloat the trusted computing base and erode cross-app isolation boundaries.

We reenvision the web interface based on the notion of a *pico-datacenter*, the client-side version of a shared server datacenter. Mutually untrusting vendors run their code on the user's computer in low-level native code containers that communicate with the outside world only via IP. Just as in the cloud datacenter, the simple semantics makes isolation tractable, yet native code gives vendors the freedom to run any software stack. Since the datacenter model is designed to be robust to malicious tenants, it is never dangerous for the user to click a link and invite a possibly-hostile party onto the client.

1 Introduction

A defining feature of the web application model is its ostensibly strong notion of isolation. On the desktop, a user use caution when installing apps, since if an app misbehaves, the consequences are unbounded. On the web, if the user clicks on a link and doesn't like what she sees, she clicks the 'close' button, and web app isolation promises that the closed app has no lasting effect on the user's experience.

Sadly, the promise of isolation is routinely broken, and so in practice, we caution users to avoid clicking on "dangerous links". Isolation fails because the web's API, responsible for application isolation, has simultaneously pursued application richness, accreting HTTP, MIME, HTML, DOM, CSS, JavaScript, JPG, PNG, Java, Flash, Silverlight, SVG, Canvas, and more. This richness introduces so much complexity that any precise specification of the web API is virtually impossible. Yet we can't hope for correct application isolation until we can specify the API's semantics. Thus, the current web API is a battle between isolation and richness, and isolation is losing.

The same battle was fought—and lost—on the desktop. The initially-simple conventional OS evolved into a rich, complex desktop API, an unmanageable disaster of complexity. Is there hope? Or do isolation (via simple specification) and richness inevitably conflict?

There is, in fact, a context in which mutually-untrusting participants interact in near-perfect autonomy, maintaining arbitrarily strong isolation in the face of evolving complexity. On the Internet, application providers, or *vendors*, run server-side applications over which they exercise total control, from the app down to the network stack, firewall, and OS. Even when vendors are tenants of a shared datacenter, each tenant autonomously controls its software stack down to the machine code, and each tenant is accessible only via IP. The strong isolation among virtualized Infrastructure-as-a-Service datacenter tenants derives not from physical separation but from the execution interface's simplicity.

This paper extends the semantics of datacenter relationships to the client's web experience. Suspending disbelief momentarily, suppose every client had ubiquitous high-performance Internet connectivity. In such a world, exploiting datacenter semantics is easy: The client is merely a *screencast* (VNC) viewer; every app runs on its vendor's servers and streams a video of its display to the client. The client bears only a few responsibilities, primarily around providing a *trusted path*, i.e., enabling the user to select which vendor to interact with and providing user input authenticity and privacy.

We can restore reality by moving the vendors' code down to the client, with the client acting as a notional *pico-datacenter*. On the client, apps enjoy fast, reliable access to the display, but the semantics of isolation remain identical to the server model: Each vendor has autonomous control over its software stack, and each vendor interacts with other vendors (remote *and* local) only through opt-in network protocols.

The pico-datacenter abstraction offers an escape from the battle between isolation and richness, by deconflating the goals into two levels of interface. The client implements the *client execution interface* (CEI), which is dedicated to isolating applications and defines how a vendor's bag of bits is interpreted by the client. Different vendors may employ, inside their isolated containers, different *developer programming interfaces* (DPIs). Today's web API is stuck in a painful battle because it conflates these goals into a single interface [11]: The API is simultaneously a collection of rich, expressive DPI functions for app developers, and also a CEI that separates vendors. The conflated result is a poor CEI that is neither simple nor well-defined. Indeed, this conflation explains why it took a decade to prevent text coloring from leaking private information [63], and why today's web allows cross-site fetches of JPGs or JavaScript but not XML [67]. The semantics of web app isolation wind through a teetering stack of rich software layers.

We deconflate the CEI and DPI by following the pico-datacenter analogy, arriving at a concrete client architecture called Embassies.[1] We pare the web CEI down to isolated native code picoprocesses [25], IP for communication beyond the process, and minimal low-level UI primitives to support the new display responsibilities identified above.

The rich DPI, on the other hand, becomes part of the web app itself, giving developers unparalleled freedom. This proposal doesn't require Alice, a web app developer, to start coding in assembly. When she writes a geotagging site, she codes against the familiar HTML, CSS, and JavaScript DPI. But, per the datacenter model, that DPI is implemented by the WebKit library [62] that Alice's client code links against, just as her server-side code links against PHP. Because Alice chooses the library, browser incompatibilities disappear.

Suppose a buffer overflow is discovered in libpng [50], a library Alice's DPI uses to draw images. Because Alice links WebKit by reference, as soon as the WebKit developers patch the bug, her client code automatically inherits the fix. Just like when Alice fixes a bug in libphp on her server, the user needn't care about this update.

Later, Alice adds a comment forum to her application. Rendering user-generated HTML has always been risky, often leading to XSS vulnerabilities [29]. But Alice hears about WebGear, a fork of WebKit, that enhances HTML with sandboxes that solve this problem robustly. DPI libraries like WebGear can innovate just as browser vendors do today, but without imposing client browser upgrades; Alice simply changes her app's linkage.

Ultimately, independent development of alternative DPIs outpace WebGear, and Alice graduates to a .NET or GTK+ stack that is more powerful, or more secure, or more elegant. Alice chooses a feature-full new framework, while Bob sticks with WebBSD, a spartan framework renowned for robustness, for his encrypted chat app. Taking the complex, rich semantics out of the CEI gives developers *more* freedom, while making cross-vendor isolation—the primary guarantee established by the client—more robust than today's web API.

Via the pico-datacenter model, we develop a CEI with:

- a minimal native execution environment,
- a minimal notion of application identity,
- a minimal primitive for persistent state,
- an IP interface for all external app communication,
- and a minimal blit-based UI semantically equivalent the screencast (VNC) model discussed above.

Such an ambitious refactoring of the web interface is necessary to finally resolve the battle between rich DPIs and a simple, well-specified CEI. While it's difficult to prove such a radical change unequivocally superior, this paper aims to demonstrate that the goal is both realistic and valuable. It makes these contributions:

- With the pico-datacenter model, we exploit the lessons of autonomous datacenter tenancy in the client environment (§3), and argue that the collateral effects of the shift are mostly harmless (§8).
- We show a small, well-defined CEI specification (§3) that admits small implementations (§6.1) and hence suggests that correct isolation is achievable.
- With a variety of rich DPI implementations running against our CEI, we demonstrate that application richness is not compromised but enhanced (§6.2).
- We show how to replace the cross-app interactions baked into today's browser with bilateral protocols (§4), maintaining familiar functionality while obeying pico-datacenter semantics.
- We implement this refactoring (§5) and show that it can achieve plausible performance (§6.3, 6.4).

2 Trends in Prior Work

Embassies is not the first attempt to improve web app isolation and richness, and indeed prior proposals improve on one or both of these axes. However, *they do not provide true datacenter-style isolation* — they incorporate, for reasons of compatibility, part or all of the aggregate web API inside their trusted computing base (TCB).

2.1 Better Browsers for the Same API

Chrome and IE8+ both shift from a single process model to one that encapsulates each tab in a separate host OS process. This increases robustness to benign failures, but these modifications don't change the web interface—multiple apps still occupy one tab, and complex cross-app interactions still occur across tabs—hence isolation among web apps is still weak. OP's browser refactoring [20] is also constrained by the web API's complex semantics.

Given this constraint, IBOS pushes the idea of refactoring the browser quite far [55]. It realizes the idea of sites as first-class OS principals [26, 57], and containerizes renderers to improve isolation. IBOS must still include HTTP to define $\langle scheme, host, port \rangle$ web principals, and must use deep-packet inspection on HTML and MIME to partially enforce the Same-Origin Policy (SOP) [67]. IBOS cannot enforce the full SOP, such as the restriction on image fetching (§3.1.4).

The Gazelle browser [58] treats sites and browser plug-ins as principals to improve isolation, but like the above systems, it maintains the existing web interface. The follow-on Service OS project [59] extended this work to encompass desktop apps, flexible web principals [45], device access, and resource management [44].

[1] An embassy is an autonomous enclave executing the will of its home country; the host territory enables multiple embassies to operate side-by-side in isolation.

All of these systems restructure the browser to improve isolation, but they are hampered by adherence to the complex web interface, in which the isolation boundary is defined in part by images, JavaScript execution [67], and fonts [4, 63]. In contrast, the pico-datacenter model imposes a new interface that makes the isolation boundary obvious and sustainable.

2.2 Changing the Web API

Many have observed that the HTML DPI isn't the best API for all web apps. An early alternative was Java [19]: a new execution and isolation model. However, because the execution model was new, and no conventional libraries worked with it, Java's CEI had to incorporate a new batch of rich interfaces and functionality, starting with the AWT GUI library. These libraries expanded the CEI (and hence the TCB), weakening the promise of isolation. The practical need for a rich DPI combined with a non-native execution model led to CEI bloat.

Atlantis [41] replaces the web's DPI with a lower-level CEI. Its executes a high-level language, and hence practical deployment of the model faces the same constraints as Java: Either it offers a limited DPI until a massive effort ports existing libraries to the new language, or it caves in and admits rich native libraries as part of the CEI (such as its `renderGUIWidget` call).

Our pico-datacenter proposal naturally evokes Tahoma [9], which defines the CEI as a hardware-compatible virtual machine. However, the Tahoma CEI isn't minimal; it includes all of HTTP and XML to specify app launch, and full hardware virtualization is needlessly broad, including x86 intricacies such as I/O ports and APICs that are irrelevant to web apps. More importantly, apps interact locally through "bins," but Tahoma doesn't explain how to use them to replace conventional web-style interactions without expanding the CEI (cf. §4), or even how to download big applications without adding a trusted cache to the CEI (cf. §3.1.3).

Various browser plug-ins, such as Flash and Silverlight, expand the existing web API to give developers options other than HTML and JavaScript. Xax [25] and Native Client [66] introduced the idea of native code web plug-ins. NaCl's SFI-based isolation requires architecture-specific reasoning, significant changes to DPI toolchains, and runtime overhead. Xax uses OS page tables, an approach that our CEI maps naturally to.

While the technologies above improve various aspects of the web, the broad approach of unioning a new interface onto the existing web API does nothing to deconflate the web's DPI and CEI and may actually introduce to security vulnerabilities [28, 60].

Mobile app platforms, such as Android, introduce an app model competitive with the web's click-anything model. But Android's permissions are closer in spirit to the desktop's model: the device and its data are sacred; installing an app explicitly welcomes the app into that sacred domain. In practice, users incorrectly trust app stores to vouch for app fidelity [31]. Our inter-application protocols (§4) evoke Android's Intents, but Embassies communication uses IP, emphasizing that a message's local origin implies *nothing* about its authority. At a low level, Android isolation is implemented with Linux user IDs [46], a subtle isolation specification wound throughout a complex kernel.

At the architectural level, our proposal employs the principle of a native, low-level interface to execution and I/O, similar to the Exokernel [30]. The Exokernel, however, aimed to expose app-specific performance opportunities; in Embassies, the low-level interface serves to maximally enforce isolation boundaries among vendors. The Exokernel project said little about how to restore inter-app functionality in a principled fashion.

3 Embassies: A Client's Pico-Datacenter

Section 1 proposed a model in which the client becomes a pico-datacenter hosting mutually distrusting apps. This section describes a specific instantiation of that idea (Fig. 1), starting with the basic execution environment offered by the pico-datacenter (§3.1). The whole reason for running an app on the client (rather than in a real data center) is proximity to the UI; to exploit this, the pico-datacenter provides each app with a minimal pixel blitting interface (for transferring pixel arrays to the screen), and the primitives needed for app-to-app display management (§3.2).

The resulting CEI (Fig. 2) has only 30 system calls, each with very simple semantics. There are no deep recesses of functionality hiding behind ioctls, making the implementation, or *client kernel,* quite small (§6.1).

3.1 Execution Environment

In the datacenter, each vendor defines its own app down to native code. Applying the pico-datacenter metaphor, our proposed CEI defines an application as a process started from a boot block of native code, running in isolation in a native environment with access to basic, microkernel-like services such as memory, synchronization and threads. An app communicates with remote servers *and* other local apps via IP packets, and it bootstraps storage from a single simple CEI call.

3.1.1 Execution: Native Code

Our client-side pico-datacenter is inspired by the success of server-side Infrastructure-as-a-Service (IaaS) systems, wherein mutually distrusting server apps occupy a shared datacenter. Server-side developers can build apps atop their choice of standard virtual machine images, or they can fine-tune or even replace the entire OS, making it easier to port existing apps. Platform-as-a-Service

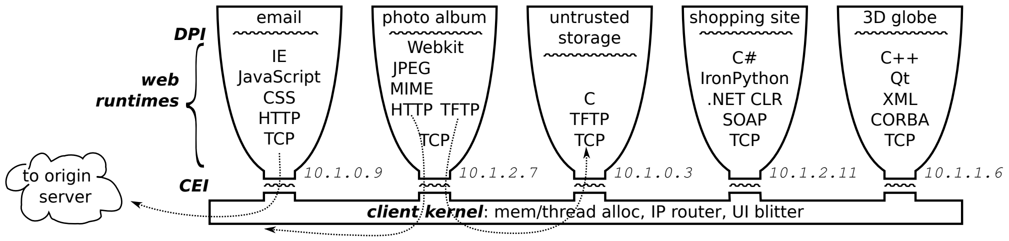

Figure 1: **The Embassies Pico-Datacenter.** *A minimal native client execution interface (**CEI**) admits a diverse set of developer programming interfaces (**DPI**s), from the web's HTML to .NET, Qt, or Gtk. Each app communicates with other servers and client apps using IP. Any protocols above IP, from TCP to HTTP to decoding a JPEG, are implemented in libraries selected by each app. Each app renders its own UI on a private framebuffer, which the **client kernel** blits to the screen (Fig. 3).*

```
allocate_memory          get_app_secret
free_memory
process_exit             accept_viewport
                         map_canvas
thread_create            update_canvas
thread_exit              receive_ui_event

futex_wait               verify_label
futex_wake
get_events               transfer_viewport
get_time                 sublet_viewport
set_clock_alarm          modify_viewport
                         repossess_viewport
get_ifconfig             tenant_changed
alloc_net_buffer
free_net_buffer          ensure_alive
send_net_buffer          endorse_me
receive_net_buffer       verify_endorsement
get_random
```

Figure 2: **The Complete Embassies CEI.** *All 30 functions are non-blocking, except `futex_wait`, which can be used to wait on events that signal the completion of long-running calls.*

(PaaS) layers elegantly and efficiently on top of IaaS. Because of the simplicity of the IaaS interface, it can clearly deliver on its promise of inter-tenant isolation. Indeed, *both* Google [18] and Microsoft [43] started with PaaS but then shifted to IaaS.

By analogy, our model allows vendors to build client-side apps atop their choice of standard DPIs, including high-level languages, or they can fine-tune or replace them as desired. This reduces the pressure to bloat the CEI with new features (§2.2), since the apps can link to new feature libraries, above the level of the CEI.

While the CEI executes native-code instructions, developers obviously won't be writing code in assembly (§1). A developer writes to a high-level DPI, and the DPI implementation emits native code, including the machinery to assemble the app from a boot block.

In particular, for web apps written against the current web DPI, the functions described in this section are hidden from the developer. These functions are used by a code module called the *web runtime* (§5.2.5), which implements the web DPI.

3.1.2 Identity: Public Keys

The pico-datacenter identifies its tenants the same way entities anywhere on the open Internet are robustly identified: by associating each process with the public key of the vendor responsible for it. In other words, Embassies's principals are public keys, so an app may consist of multiple processes running different code, but Embassies will treat them all as a single principal.

Embassies identifies the principal for a process during process start. Each process starts from a self-contained, native-code boot block (§3.1.1). That boot block is signed by a private pair held by the process' principal. Before a process starts, the client kernel checks the signature, and henceforth it associates the new process with the corresponding public key; §3.2 discusses how this identity is conveyed to the user.

The CEI does not specify how the signed boot block is acquired, leaving it up to the DPIs to define and evolve suitable mechanisms — see §4.1 for an example.

The CEI also takes a data-center-based approach to handling app instances, i.e., multiple processes that belong to the same principal (i.e., public key). When a customer contacts a data-center tenant, e.g., Netflix, she contacts the vendor, rather than directly specifying a particular virtual machine running a particular binary. Similarly, with Embassies, the CEI does not specify how to contact a specific process belonging to a principal. Instead, each app vendor can choose to make all of its processes available for communication, or the vendor may choose to use one process to dispatch requests to other processes it controls.

This minimal notion of app identity contrasts with today's web, which distinguishes principals based on the protocol, host, and port used to fetch the app; thus the very specification of app identity incorporates the complexity of TCP, HTTP, HTTPS, and MIME.

Embassies's minimal definition provides a strong notion of identity, making it simple to determine when a message speaks for an application and to enable secure communication amongst apps (§3.1.4). Many awkward consequences of the web's cobbled-together definition vanish [27]; today a vendor may own two domain names but cannot treat them as one principal, or a single domain may represent multiple entities (e.g., GeoCities or MySpace) but is treated as one principal.

However, defining and verifying app identity on an end user's client is more challenging than for a remote server, because it is not safe to download a vendor's private key to a client. For instance, Flickr uses its private key to authenticate its server, but it would never embed that key in the code it downloads to a client.

Our solution is based on the observation that, after verifying a vendor's signature on a binary, the client kernel can authoritatively state that the app *speaks for* [35] that vendor *on this machine*. The `endorse_me` call allows an app to obtain such a certification for a crypto key it generates, and other apps on the local machine can verify this with `verify_endorsement`, similar to authentication in the Nexus OS [54]. Since local apps already depend on the client kernel for correctness and security, this introduces no new dependencies.

Endorsing apps via crypto keeps the client kernel simple and makes explicit the guarantees the return value provides. It also further emphasizes the pedagogical point that each app should treat communications with local apps with as much suspicion as it would treat communications with remote apps.

3.1.3 Persistent State: Pseudorandom Keys

The current web interface specifies several local storage services as part of the CEI: an object cache, cookies, and local storage. Each service must be correct to preserve app isolation; for instance, the cache can violate an app's security or correctness if it misidentifies the origin of an object. Worse, these services have complex semantics apps cannot control; for example, the browser delivers cookies on one app's behalf when a different app makes certain requests; flaws in this design lead to Cross-Site Request Forgery (CSRF) vulnerabilities [6].

By contrast, in a shared data center, apps cannot even assume the presence of local storage, let alone complex storage APIs for caches or cookies. Instead, the app's developer uses a remote storage service, such as Amazon's S3 or Azure Storage. Even if she trusts Amazon, a sensible developer uses SSL to connect to the storage service, and a less trusting developer can use additional cryptography to avoid trusting Amazon.

Hence, following the pico-datacenter analogy, our CEI does not provide any storage services directly. Instead, apps bootstrap all of their storage needs via the `get_app_secret` call, which returns a secret specific to both the app's identity and the client machine.

The app secret is stable, so when the app restarts later, it gets the same secret. An app library can use the app secret as key material to build encrypted and authenticated storage from any untrusted external store, such as a daemon on the local client machine, a server-based cloud service, or even a peer-to-peer service. Apps use this secure storage facility to save cookies and other app-specific state.

In addition, mutually-distrusting apps can share an untrusted store that acts as a common content cache (§5.2.2); each app independently authenticates (e.g., via a MAC with the app secret as a key) the cache's content.

In both cases, replay or rollback attacks can be prevented via standard techniques [38, 48].

Our client kernel implements this interface by storing a symmetric key for a pseudorandom function (AES). It applies the function to the hash of the app's public key to generate a secret unique to the $(app, host)$ pair.

3.1.4 External Interface: IP Only

Today's web API supplies an ever-expanding set of communication primitives, including content retrieval via HTML `src` attributes, form submissions, links, JavaScript XMLHttpRequests, PostMessage, and WebSockets. Each expands the complexity of the CEI.

In contrast, our pico-datacenter follows the communication model of Internet servers: It offers only IP, with simple best-effort, non-private, non-authenticated semantics. Using IP even for messages traveling on the same machine sounds slow and counterintuitive. However, it imitates the physical constraints that guided the evolution of robust inter-server protocols. Servers communicate only by value, not by mapping shared address spaces; such decoupling leaves room to design robust protocols and select robust implementations. We can keep IP's semantics while exposing good performance by supporting bulk transfer with IPv6 jumbo frames, and by exposing a zero-copy packet interface (§5).

In practice, the client kernel assigns each app an IPv6 address and a NATed IPv4 address. The client kernel's responsibility is that of any other Internet router: best-effort delivery, with no particular guarantees on integrity or privacy.

As with any other Internet interaction, to communicate securely with other parties, an app uses cryptography. For example, the app might include a server's public key, or a public key for the root of a PKI, and then communicate with the server over SSL. The CEI does not provide cryptographic operations; the app must incorporate (e.g., via a library) any crypto code it needs. However, the CEI's `get_random` call provides a supply of secure randomness for seeding cryptographic operations, like nonce or key generation.

Communicating with Remote Servers.

In today's web, communication with remote servers is deeply complicated by the web's breathtakingly ambiguous Same Origin Policy (SOP), which refers to an ad-hoc collection of browser behaviors that attempt to selectively isolate sites from one another [67].

Locally, the SOP prevents most but not all DOM-based interactions; following the pico-datacenter metaphor, Embassies enforces a stronger, simpler policy: strictly isolate apps, with interactions only via IP.

When communicating with remote servers, the SOP primarily affects when the browser attaches cookies to an outbound request, and when a webpage can fetch content from a remote server. We discard the restrictions on cookies, since in Embassies, each app, via its DPI, governs access to its own cookies and decides when to include them in a request (§4.2). The CEI never adds ambient authority [23] to an app's communications.

The SOP's restrictions on fetching remote content aren't so easily dismissed. Since a web client may be running behind a firewall, allowing untrusted apps to freely use its network connection creates a confused-deputy vulnerability [23]. For example, an evil app on a user's web client may request content from the internal corporate payroll server, which the server allows because the request originates behind the firewall. The SOP addresses this with complicated rules such as allowing an app to retrieve an image from any site and display it, but not examine its pixels. Such rules require reasoning at a high level to know that a retrieved file *is* an image.

We observe that a much simpler policy addresses the confused-deputy threat. The threat arises from allowing untrusted apps to inherit the web client's privileged position on the network; thus, we disallow that privilege. In Embassies, every app receives, either via IT network configuration or via an explicit proxy, an IP connection logically outside any firewall. We call this "coffee-shop networking" (CSN), since apps use an IP connection semantically equivalent to a public network, e.g., in a coffee shop. An app that accesses enterprise resources can include a VPN library. To avoid asking the user to authenticate more than once, the app may choose to share its VPN connection with other enterprise-approved apps that it authenticates cryptographically (§3.1.2).

In fact, the necessary environment for CSN is emerging due to the "consumerization of IT" [47], which encourages institutions to make logically-external connections available for untrusted devices and to harden internal servers. Windows 8 grants apps an "internetClientServer" permission, a policy equivalent to CSN. [42]

We discuss the potential for resource abuse (e.g., Denial-of-Service) in §7.

Communicating with Local Applications.

In the pico-datacenter, a local app is just another server sitting on the network, and thus intra-client communication, just as app-to-server communication, is simply IP. This keeps the CEI simple and encourages defensive app design; local apps appear no different than servers because they are no more trustworthy than servers.

However, communicating with local apps differs from servers in a crucial aspect: It is reasonable to assume that server processes are available; `map.com` can send a message to `flickr.com` and reasonably expect a running process to receive it. In contrast, a web app cannot safely assume any other app is currently running on the local client.

Thus, the CEI provides the call `ensure_alive` to ensure a local process is indeed alive locally. We deliberately make the call's semantics minimal, leaving most of the work to the calling and target apps. The calling app must somehow locate the target app's binary boot block, signed by the target app's vendor, and pass it to `ensure_alive`. If no instance of the target app (as identified by the public key that signed the boot block) is yet running, the client kernel verifies the signature, starts a container for the new app, and associates the vendor's key with the container. Thereafter, the caller app can communicate with the target app by IP, for instance to pass parameters to the second app.

Note how the `ensure_alive` primitive contrasts with a conventional OS process start: no parameters, environment, handles, or library paths. A single vendor can use `ensure_alive` to create multiple processes, which may be helpful for benign fault isolation, but because each such process shares a common principal (the vendor key of §3.1.2), there is no security isolation between such processes.

3.2 UI and Display Management

The preceding subsections carve up the client machine into a fairly standard "shared datacenter"; however, a pico-datacenter is interesting because it lives near the user. Hence, unlike a traditional datacenter, we must also specify how apps access the user interface, and how the CEI handles display management. Our guiding principle is to reason about how remote, screencast apps (§1) might coordinate to manage a dumb client's UI.

User Interface. Today's web apps specify user interfaces via a complex amalgam of HTML, CSS, JavaScript, DOM, and many other standards. Our goal of a minimal CEI drives us to the leanest feasible interface: An app may accept a rectangular *viewport* region (`accept_viewport`) and map a *canvas* into its address space (`map_canvas`) – see Figure 3. This allows the client kernel to place it in a region of memory where blitting is cheap; if the viewport is resized, another call to `map_canvas` recreates a matching framebuffer. After painting pixels onto the canvas using the rendering stack it prefers, the app asks the UI (via `update_canvas`) to blit the pixels onto the visible part of the app's viewport. When the user's input focus is in the viewport, the client kernel delivers mouse and keystroke events to the app (`receive_ui_event`).

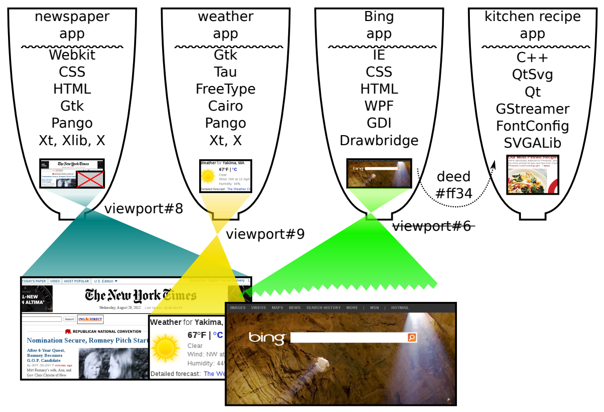

Figure 3: **UI Management.** *`Sublet_viewport` lets the newspaper nest the weather app's display inside its region. On the right, the user clicked a link on the Bing app, which used `transfer_viewport` to convert its viewport (access to the screen) into a **deed** (a secret capability), and sent the deed in a message to the kitchen app. The kitchen app will use `accept_viewport` to redeem the deed for its own viewport.*

As with the choice of native code, this refactors rich UI features into the apps, simplifying the CEI while enabling virtually any UI a DPI-developer can imagine (we discuss GPUs in §8). Indeed, because Embassies executes native code, we can employ a variety of mature UI stacks (§5.2) as DPI-supported UIs for web apps.

The client kernel labels app windows with the app's identity, so the user can select a window and know which app he is communicating with. The CEI does not use cryptographic keys directly as labels, because such keys are difficult for users to interpret. Instead, the CEI maps keys to hierarchical DNS-style labels (e.g., `bing.com`), based on and compatible with the DNSSEC PKI[2]. Before an app can accept a viewport (and hence appear on screen), the app must gather a certificate chain authenticating its label and call `verify_label`.

Naming, labeling, and visual ambiguity are hard problems; users manage to ignore most cues [52]. Our client kernel provides the minimal facility described above to address this problem, consistent with the best known methods [16, 53, 65], but we recognize that progress on this problem [10] may require CEI evolution.

Display Management. Much of today's browser functionality, such as linking, embedding, navigation, history, and tabs, are basically mechanisms for display management. To adhere to the remote screencasting abstraction (§1), we designed a viewport-management interface with capability semantics. This interface has five calls and primitive semantics; the rich browser-like functionality is built up by apps themselves (§4).

[2]Experience with SSL/TLS illustrates that deploying a large-scale PKI is challenging. Security is undermined by hundreds of certificate authorities baked into common browsers. Thus, we choose a DNSSEC-style PKI with few trust anchors and scoped naming authority.

Our CEI supports the transfer of a viewport from one app to another via `transfer_viewport`, which accepts a viewport and returns a *deed*, a secret capability that can be passed to another app via a network message. The receiving app can call `accept_viewport` to redeem the deed for a viewport it can draw in. Transforming a viewport into a deed destroys the viewport, and accepting a deed into a viewport destroys the deed; thus only one app has access to a viewport at a time.

Rather than transfer an entire viewport, an app may wish to delegate control over a rectangular sub-region of its viewport via `sublet_viewport`. This creates a deed that can be passed to another app. It also yields a handle to the sublet region, with which the parent app can resize or move the region via `modify_viewport`, or revoke it with `repossess_viewport`.

To allow communication (e.g., changes in viewport size) between the app that sublets a viewport (the *landlord*) and the app that accepts it (the *tenant*), our CEI provides each landlord-tenant pair with a fresh symmetric key that can be used to authenticate and optionally encrypt viewport-related communication. Since the key provides secrecy, integrity, and authenticity, apps may use anonymous communication mechanisms (e.g., anonymous broadcast from a random IP address) to better protect the user's privacy.

4 Refactoring Browser Interactions

§3 introduced a CEI with minimal support for hosting pico-datacenter apps and enabling them to share the UI. This section shows how we can build up equivalent functionality *inside the apps* to restore the rich cross-app interactions familiar in the classic browser. Less browser-specific interactions, such as copy-and-paste, can be handled via techniques from related work (e.g., [51]).

Rather than bake these rich interactions into the client, each interaction is reconstructed as a bilateral protocol between cooperating apps. This refactoring gives application vendors the autonomy to make security/functionality tradeoffs, for example by choosing a more robust implementation of a given protocol, implementing only a subset of it, or even refusing it altogether.

More importantly, refactoring interactions as protocols clarifies the underlying semantics, whereas in today's web, complex feature interactions lead to surprising security implications. For example, refactoring provides new perspective on Cross-Site Request Forgery (CSRF) (§4.2) and policies for visited-link coloring (§4.5).

4.1 Linking

When a classic web app includes a link to another app, it is prepared to transfer control of its screen real estate in response to the user's click. In the current web API, the hyperlink is a high-level function, bundling name resolution, app fetch, app start, app window labeling, parameter passing, cookie transmission, and screen real-estate transfer into a single browser feature. In contrast, the pico-datacenter model partitions these tasks mostly between the app that contains the link and the app being linked to; the client kernel provides minimal support.

Consider `caller.net`, an Embassies app written in a classic HTML DPI, containing a hyperlink:

`<a href="target.org/foo?x=5&y=10">`

When a user clicks the link, the caller app identifies and contacts the target app. First, it translates `target.org` into a strong identity, perhaps by resolving it, via DNS or some stronger PKI, into a public key for the target app (§3.1.2) — §8 discusses legacy servers. Second, it contacts a local instance of the target app via local broadcast.

Since the target app may not be running locally, the caller uses `ensure_alive` (§3.1.4) to ensure that the target app has a presence on the client (in the local pico-datacenter). This requires caller.net to fetch a signed boot block matching the web runtime's ISA; it finds it as it found `target.org`'s public key. Target.org's tiny bootstrap executable retrieves and verifies the rest of its code and data, by its own means. Once `target.org`'s web runtime calls `verify_label` (§3.2), the vendor has a presence on the client.

From its client presence, `target.org` responds to caller.net's broadcast via unicast IP. The two web runtimes have their public keys endorsed by the client kernel (§3.1.2), and use them to create a secure communication channel. Caller.net's web runtime then transforms its viewport into a deed (§3.2), and sends a message to `target.org` containing the deed and the entry point parameter `/foo?x=5&y=10`. If `target.org` wishes to pass the request to its server, it does so itself (§4.2); the client kernel has no notion of HTTP. If `target.org` wishes to include a client-stored cookie, it fetches and forwards its own cookies (§3.1.3); the client kernel has no notion of HTTP cookies.

While the above process may sound heavyweight, much of it is simply a refactoring of the work done today by the browser. Furthermore, our results (§6.3) show that the overhead of app start is quite reasonable.

4.2 Cross-Domain Communication

Today's web offers many communication mechanisms, such as XMLHttpRequest, script and image inclusion, PostMessage, and third-party cookies. Refactoring them into explicit app-implemented protocols is easy.

XMLHttpRequest and HTML `script` and `image` tags use app libraries that employ TCP, HTTP, and XML libraries to reproduce standard functionality internal to the app, relying on the CEI only for IP (§3.1.4). The simplicity stems from Embassies's handling of confused-deputy problems at the IP level (§3.1.4).

PostMessage lets one local client app send messages to another. In Embassies, these messages simply become IP packets, optionally protected cryptographically.

Automatic HTML cookie semantics mixed with imperative code lead to cross-site scripting vulnerabilities; the HttpOnly attribute attempts to curtail the complexity enough to mitigate the threat [5]. In Embassies, an app can only manipulate a cookie belonging a separate vendor via an explicit IP request to the cookie's owner. The owner enforces policies on which cookies are exposed and to whom.

This refactoring reveals how CSRF threats can now be addressed by individual vendors. CSRF occurs when a malicious app dupes the browser into sending a request to a valuable app's server that's indistinguishable from a legitimate request: It looks like the user submitted a form, and it contains the valuable app's cookies. In the refactored relationship, it is straightforward for the valuable app to implement separate mechanisms for its user interactions versus its invocations from other apps.

4.3 Embedding

Visually embedding another app, such as in an `iframe`, is just like navigation, except the landlord uses `sublet_viewport` rather than `transfer_viewport`. When a sublet viewport is transferred to another app, three parties cooperate in the transfer: the old tenant, the new tenant, and the landlord. At the conclusion of the transfer, the new tenant but not the old tenant has access to the viewport, and the new tenant can communicate with the landlord without revealing its identity. The parties achieve this with a three-way protocol that performs an atomic transfer. A failed party can violate liveness, but the landlord can recover after a timeout with `repossess_viewport`.

4.4 Favorites

Classic browsers allow the user to bookmark favorite pages. This interaction becomes a protocol in Embassies: One client app acts as the user's bookmark repository. A user gesture tells an app to send a bookmark to the repository, consisting of the app's identity and an opaque entry-point parameter the app can use to reconstruct the user's state. This refactoring makes it clear that the repository gets to know which vendors the user has explicitly bookmarked, and nothing more.

4.5 Navigation Threading and History

A classic web browser tracks the user's history, enabling different views of the link graph the user traversed: the *back* button walks a path in the graph, *history* records the graph's nodes (i.e., sites the user visited), and *link coloring* displays the nodes via the current app's outbound links.

One could implement these functions in an Embassies ecosystem by declaring a trusted repository app, and adding to the linking protocol (§4.1) a step that submits a "bookmark" for the linked page to the repository.

Such a refactoring indicates that the repository is entrusted with quite a trove of private data. Furthermore, implementing link coloring reveals the repository's knowledge to every app. One could band-aid the damage by having the repository render links as embedded displays (§4.3) on behalf of apps, to avoid revealing the node graph to adversarial apps. This is essentially how the classic browser, which acts a trusted history repository, protects user privacy. Achieving privacy has been a long, complex battle [4]. In Embassies, such a relationship is at least well-defined.

However, we find the relationship too promiscuous. Instead, we deliberately abandon global history. For link coloring, we accept downgraded behavior, leaving individual applications to record their own outgoing clicks. For example, Bing can remember which links you have clicked on *from Bing*, and color such links purple. If you've arrived at `embarrassing.com` via some other path, but never from Bing, then the link to that site remains blue on Bing's results page. This provides weaker semantics than the classic web, coloring links as edges rather than nodes, but has simple privacy implications.

The back button requires each app only to know its local neighborhood of the graph. An app can provide internal navigation itself. To span apps, the linking protocol (§4.1) is extended to carry an app identity and an opaque blob, a "bookmark" for the reverse edge. When the user backs out of the target app, the target invokes the bookmark with the linking protocol to replace its display with the prior app. This allows an app to cause the back button to go to unexpected sites, break, or vanish entirely. In the classic web, the complexity of redirects and automatic navigation can cause similar mischief, rendering the browser's back button similarly problematic.

This scheme reveals the identity of the caller app to the target app, just as Referrer headers do today. The alternatives are to have a trusted, centralized store of the navigation graph (the classic browser's behavior, an approach we dislike), or to let apps create anonymous proxy identities to hide their identity from those they link to.

4.6 Window Management and Tabs

Managing overlapping windows or tabs is achieved using the same primitives that manage sublet viewports (§3.2). Thus an ordinary application, typically the first one Embassies starts, provides window resizing handles and tabs, treating the enclosed content as embedded iframes (§4.3). As with any such UI relationship in Embassies, the window manager cannot violate the privacy or integrity of the apps whose windows it manages.

The landlord controls the z-order of its tenants (presently unimplemented). The client kernel provides no support for transparency; if separate apps wish to implement it, they must expose their pixels to some app they trust to implement the blending.

5 Implementation

To evaluate the minimality and simplicity of the CEI, we implement three instantiations (§5.1). To evaluate the richness offered to developers, we port three full DPIs to Embassies (§5.2). All the code is available [1].

5.1 CEI

We have built a complete CEI implementation for Linux and a nearly complete one for the L4 microkernel [24]. For debugging purposes, we built, but omit for space, a complete non-isolating Linux implementation.

5.1.1 The Linux KVM Monitor

The measurements in §6 all run on our `linux_kvm` monitor, which relies on Linux KVM [32] to provide a virtual CPU for each app. For memory, the client kernel allocates a large contiguous block of virtual memory, and gives pieces of it to the app in response to memory requests. The client kernel performs thread scheduling, and it maintains a table of futex queues to block app threads performing `futex_wait`. It also directly implements the clock, timer, and crypto primitives.

A single central coordination process manages a connection to an X display, our UI mechanism. It also implements a logical IP subnet for routing packets between apps and to the Internet. Each app communicates with the coordinator using sockets. To connect to the Internet, the coordinator injects and intercepts packets at the IP layer using `tun`. To provide NAT, it employs the iptables functionality built into the Linux IP router. When a client is behind a firewall, it routes packets over an IP tunnel to a CSN proxy. For performance when moving large data between apps, it provides a zero-copy path for IPv6 jumbo frames, using shared memory.

5.1.2 The L4/Genode Monitor

We have also implemented the CEI on an L4::Pistachio microkernel [24], building on the Genode OS [14, 17] framework's memory allocation, RPC abstractions, and Nitpicker UI [15]. It runs all of the rich-DPI applications the Linux KVM monitor does.

5.1.3 Alternatives

While the linux_kvm monitor depends on hardware virtualization, the CEI doesn't require it. It supports any computer with an MMU [25], perhaps using OS mechanisms like seccomp [36] or PTRACE_SYSEMU.

5.2 DPIs

We have linked three full DPIs against Embassies: classic web, Gnome/Gtk, and KDE/Qt. The classic web

DPI is built from a Webkit-based [62] browser, Midori [56], which is itself built on Gtk libraries. The KDE/Qt toolkit is almost entirely distinct, but it shares its bottom layers (X, libc) with Gtk. In addition, we built a minimal DPI (§5.2.1) that runs native C code and accesses CEI facilities directly. Each DPI is a stack of software that talks to the CEI at the bottom layer.

5.2.1 POSIX Emulation

Embassies's POSIX emulation layer (EPE) lies at the bottom of each DPI we implemented. It supports the POSIX-facing libc, which in turn supports Gtk and Qt. For instance, libc implements its malloc function by calling `brk` or `mmap`, and EPE converts these into an `allocate_memory` call to our CEI.

Because POSIX identifies system resources via the filesystem namespace, EPE includes a virtual in-process filesystem (VFS) implementation, with several underlying filesystems. Implementing facilities as VFSs is often easier than modifying app logic in higher layers [25].

5.2.2 Virtual Filesystems

EPE includes a read-only filesystem that holds an image of the applications' executable and data files. EPE also contains entry-point code, which maps a copy of the dynamic loader `ld` and calls it with the path to the app executable in the read-only filesystem.

This read-only filesystem accesses data from a storage service (§3.1.3) via an FTP-like protocol. Files are identified by their hash values, which are computed using Merkle trees [40] to facilitate content-based block sharing with other apps. If the service doesn't have a requested block, the read-only filesystem contacts the app's origin server. Fetching files incurs costly round trips, so the read-only filesystem initially prefetches a tar-file of the app's startup files. Requests that fail in the tar-file fall through to individual cache requests.

To store an app's temporary files, EPE provides a RAM-disk VFS. For intra-app communication, EPE provides access to pipes and sockets via another VFS. EPE translates app reads from `/dev/random` into `get_random` CEI calls. Reads from `/proc` are partially emulated within EPE, e.g., to provide the stack layout to garbage-collection libraries. A VFS provides a filesystem for securely storing persistent data (§3.1.3), e.g., cookies; these employ a local storage service. Another VFS provides access to a server-side store.

5.2.3 Xvnc

All our DPIs are currently based on X graphics. Our implementation satisfies X requests via a modified Xvnc library. Xvnc speaks the X protocol at the top and the VNC remote-frame-buffer protocol at the bottom. We replace the bottom with code that uses our CEI's viewport/canvas instead. This modified about 350 SLoC.

5.2.4 Gtk and Qt

Once these layers are in place, getting a much richer toolkit in place is surprisingly straightforward, even though these toolkits consist of 50–100 libraries. Some Gnome-based applications were insistent that a Dbus object broker be present; we satisfy them by simply spinning one up within the app. Other apps, such as Gimp, draw numerous toolboxes. We load a twm window manager alongside Gimp to enable the user to manipulate the toolboxes on a single Embassies viewport.

5.2.5 libwebkit and Midori

For our HTML DPI, we started with Midori [56], a browser based on the libwebkit HTML DOM implementation [62]. Midori and Webkit are in turn Gtk apps, so most of their requirements are satisfied by the techniques above. We implemented a tab manager (§4.6) and inserted hooks in Webkit's link, GET, and iframe mechanisms to connect them to the linking (§4.1), navigation (§4.5), and embedding (§4.3) protocols. For example, in the link case, the hook retrieves the tenant viewport from Xvnc, converts it into a deed, and forwards it to the destination app. We have not yet implemented window management, favorites, or history management, though these should be straightforward, since window management is a subset of tab management, and favorites and history are handled by normal apps.

5.2.6 Alternative DPIs

Drawbridge ports Windows and .NET to a "picoprocess" interface close to our CEI, making it a good candidate for a web DPI [49].

5.3 Architectures

We have only implemented an x86-32 variant of the CEI. Nothing in the CEI depends on the ISA; other architectures would be straightforward. The x86 CEI variant inherits an ISA quirk: all popular x86 software frameworks abuse an x86 segment register as a thread-local store pointer to reduce pressure on the paltry x86 register file. We support this by adding a `x86_set_segment` call to the x86 CEI variant. The call has trivial semantics and no security impact; supporting it lets most library binaries run unmodified, greatly easing porting effort.

6 Evaluation

This evaluation answers four questions: Does the CEI achieve its goal of minimality (§6.1)? Does it support diverse, rich DPIs (§6.2)? We shift the burden for application bootstrapping onto apps themselves; how big is the performance cost (§6.3)? When each app brings its own DPI, is the memory burden acceptable (§6.4)?

We test with an HP z420 workstation with a four-core, 3.6GHz Intel Xeon E5-1620 CPU and 4GB of RAM.

Client Kernel	SLoC	Underlying TCB
linux_kvm	28,138	Linux (millions)
linux_dbg	21,445	Linux (millions)
bare_iron	16,714	Genode, L4 (~70K)
Firefox	4,561,642	Linux (millions)
Chrome	6,722,375	Linux (millions)

Figure 4: **TCB.** *Unlike today's web API, the Embassies CEI admits modest implementations.*

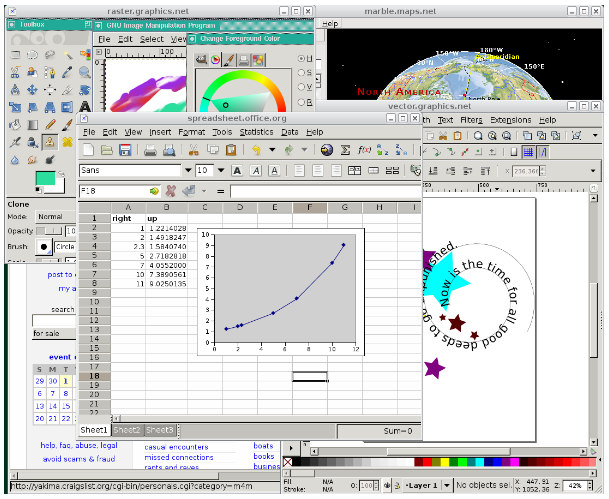

Figure 5: **Diverse DPIs.** *Native code as CEI enables diverse DPIs. This screenshot shows apps Craigslist (Webkit/HTML), Gimp (Gtk), Marble (KDE/Qt), Inkscape (Gtk), and Gnumeric (Gtk) running on the Embassies CEI. Not shown are Abiword (Gtk), Gnucash (Gtk), or Hyperoid (EPE).*

6.1 Minimality/Simplicity of the CEI

CEI minimality both improves isolation by reducing TCB size, and leaves richness up to the app's libraries. Figure 4 counts the client-kernel code sizes [64], which represents the amount of code all apps must trust. Each CEI implementation depends on some underlying OS. Although Linux is huge, CEI safety depends only on a subset of its semantics, memory management and the `kvm` driver. Likewise, the display uses X, but only pixel rectangles, not X's security model. The L4 implementation further supports the hypothesis that the Embassies CEI can be implemented with relatively little code.

Any application running *on* the CEI may include millions of lines of code, but the vendor controls *which* lines, and none of this code increases the TCB of any other app.

6.2 Diversity of DPIs

We have demonstrated half a dozen applications running on three major DPIs—Gtk, Qt, and Webkit—comprising 143 MB of binary in 200 libraries (Fig. 5).

6.3 Performance

We consider it worthwhile to spend some performance for a richer, more secure web. How much performance are we spending?

CPU Overhead. We ran a subset of the SunSpider JavaScript benchmark [61] on both Linux and Embassies. We also ran Gimp image rotations as a native macrobenchmark. Unsurprisingly, in both cases the difference is negligible: results are within 2% with standard deviations of 1%. These results confirm that a well-designed, low-level CEI need not add any additional CPU overhead to such computations.

Communication. To evaluate the overhead of IP communication between local apps, we measured the time Midori takes to fetch its cookies from an untrusted store (§3.1.3). This involves not only IP latency, but the cryptographic overhead of decrypting and verifying the integrity of the data. Nonetheless, we find that Midori can read or write a cookie in under a millisecond; refactoring interactions into protocols adds negligible overhead.

As discussed below, we use zero-copy data transfers and caching to reduce the overhead of transferring large amounts of data (e.g., DPI images) between apps.

App Start. The most significant impact of our refactoring is that, rather than intimately sharing a monolithic browser's heap, each app bootstraps its own DPI layers. How much does this increase the latency between when a user clicks a link and when the app launches?

The very first time the client ever encounters a new DPI, she must, of course, download it, just as she would if she selected a new browser. Subsequently, the DPI's files can be served rapidly out of a local, untrusted cache (E-Hot in Figure 6). Indeed, clever caches will likely preload popular DPIs to avoid even the first-time download. In a "patched" start (E-Patch, Fig. 6), the app's image is absent from the cache, but another app based on a similar DPI is present, and the Merkle tree reveals that only a delta is needed (§5.2.2). Thus, deviation from popular DPIs will result in an initial app load time proportional to the amount of deviation. One reason a vendor might deviate from a popular DPI is to fix a broken library. For example, libpng patched an overflow vulnerability in February 2012 [50]. In this case, the "patched" Midori is 76MB but differs from the cached Midori only by the 0.5MB repaired libpng library. Once the delta has been fetched, subsequent fetches by any other vendor using the patched libpng also hits the cache.

To reduce bootstrap time, we start each app from a tar file, so the entire image is transferred from the untrusted cache in one packet (§5.2.2), reducing overhead and enabling zero-copy optimizations. The first time an app runs, its loader verifies the hash (SHA-1) of the tar file; to save time on future loads, the app uses its platform-specific secret key (§3.1.3) to MAC the tar file and stores the MAC value in untrusted storage. MACs such as VMAC [33] can be verified faster than a hash.

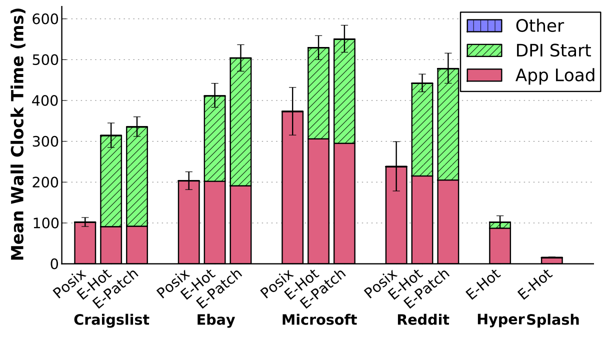

Figure 6: **App Startup Latency.** *Four web apps, a native game (Hyperoid), and a splash screen. For the web apps "App Load" is the time to fetch and render the HTML content. Error bars show standard deviations of total time over 10 runs.*

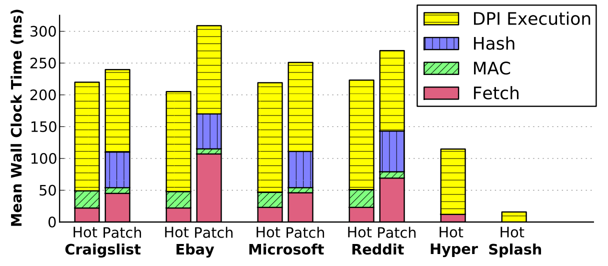

Figure 7: **DPI Start Breakdown.** *Fetching the DPI from the cache costs more in the warm case, due to fetching upstream blocks and the need to hash rather than MAC for integrity. Mean of 10 trials.*

Figure 6 assumes zero network delay to avoid burying Embassies's overheads in high network latencies. Our untrusted cache only supports UDP, incurring many RTTs hidden by this zero-delay assumption, but in deployment, it would pipeline blocks with TCP, incurring RTTs typical of HTTP transfers.

We load a set of popular websites in Midori on Linux, which takes 102–373 ms. In contrast, a hot start on Embassies takes 314–529 ms, and a patched start takes 335–551 ms. Unsurprisingly, the app load (i.e., web page fetch and render) step is similar in both cases. Embassies's overhead comes primarily from the need to fetch, verify, and boot the Midori DPI. Most of that time (Fig.7) comes from starting Midori from scratch, which even on Linux requires 130 ms ($\sigma = 7$). This is unsurprising, since Midori app starts are assumed rare, and hence unoptimized. This overhead could be mitigated by checkpointing to avoid library relocation [12, 39], by applying Midori-specific tuning (e.g., not loading every available font on startup), or by displaying a splash screen until the app achieves interactivity. Figure 6 shows that Embassies can display such a splash screen (1.5MB) in 15 ms ($\sigma = 1$). As an example of optimized start time, we ported a game, Hyperoid, to Embassies. It starts in 102 ms ($\sigma = 15$) when cached.

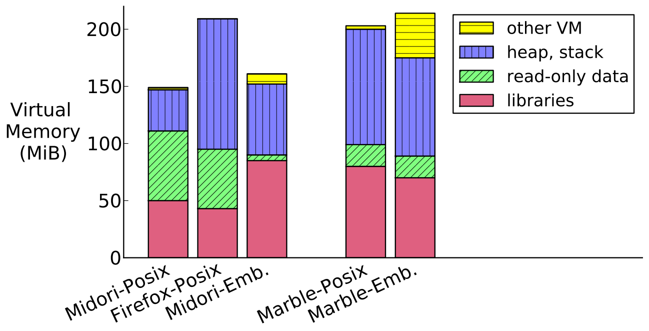

Figure 8: **Memory.** *Embassies DPI implementations consume virtual memory comparable to their POSIX progenitors.*

These costs are within the ballpark of a page load, but further improvements are possible. A hot app can remain resident to avoid a start altogether. The tar file is captured at file granularity, but many files are barely touched; page granularity would reduce the 76MB image to 33MB.

In summary, while the 177–300 ms overhead of our prototype is a non-trivial delay, there are plenty of opportunities to improve it; our refactoring makes those opportunities accessible to vendors. Overall, we are glad to exchange the challenges of security and app richness for the ordinary task of systems performance tweaking.

6.4 Memory Usage

If every vendor's application loads its own copy of a DPI implementation, will memory usage be overwhelming? Prior work shows that this style of statically linked code need not cost significantly more memory than traditional shared code implementations [8, 22].

Figure 8 contrasts virtual memory usage of POSIX implementations with those in Embassies. Since it incorporates the Xvnc rasterizer and other libraries, Midori in Embassies uses 12MB (8%) more virtual memory than its POSIX equivalent. Another DPI instance, Marble running on Qt, shows similar growth, 11MB (5%).

In a conventional browser, one instance of the browser serves many applications, amortizing fixed costs of both libraries and some heap structures. The zero-copy IP router in Embassies affords the same opportunity for libraries—the untrusted cache could send the same payload to multiple applications—but our prototype does not yet implement copy-on-write. With regard to the heap, more modern browsers (IE9 and Chrome) launch one process-per-tab, creating more heaps; in Embassies, process-per-app incurs additional heap costs.

7 Security Analysis

Embassies improves security by specifying such a small, simple CEI that implementations thereof stand a reasonable chance of truly fulfilling the web's promise of app isolation. The client kernel's small TCB (§6.1) means that the amount of code all apps must trust is tiny, and hence each vendor can independently choose the right tradeoff between complex functionality and security. A gaming app can use a rich, full-featured DPI, while a banking app may choose a conservative DPI enhanced with the latest security protections. One app's insecurity never undermines the security of other apps. Finally, since the pico-datacenter model deconflates the CEI from the DPI, Embassies provides an ecosystem that resists pressure to expand the CEI, since developers can achieve arbitrary richness inside their picoprocesses.

In contrast, in today's web, many corporations still run Internet Explorer 6 for the sake of a single business-critical app. This compromise endangers all other apps on the client and the client system itself. In Embassies, the business app uses the Internet Explorer 6 DPI, which is no more (or less) dangerous to the client or her apps than a website that uses an old server-side library.

In addition to ecosystem-wide improvements, Embassies's design addresses specific web threats.

Cross-Site Request Forgery (CSRF). Today's CSRF attacks rely on the adversary's ability to trick the browser into sending out an app's cookies inappropriately (§4.2). The Embassies CEI never adds ambient authority [23] to an app's communications, so a banking app need never fear that the browser will blindly hand out its cookies.

Cross-Site Scripting (XSS). XSS flaws spring from poor library interfaces that fail to starkly distinguish data from the code that contains it; they are a client-side equivalent of server-side SQL-injection flaws. Embassies enables the vendor to migrate to rendering libraries that safely encapsulate tainted input, just as smart vendors use SQL libraries that safely encapsulate tainted input in `WHERE` clauses.

Clickjacking. Embassies resists clickjacking in the spatial domain by ensuring that each display region belongs to one viewport managed by only one app (§3.2). Vendors concerned about clickjacking in the temporal domain can implement client-side defenses, e.g., by ignoring inputs until 200ms after painting the display.

Side Channels. As in modern data centers, Embassies's pico-datacenter does not take steps to prevent side channels; i.e., one vendor may be able to infer another vendor's presence from the kernel's scheduling decisions or shared cache effects [2]. Current browsers face the same threats. Reducing the web's security problems to the existence of such side channels would be valuable progress.

Hosted Denial-of-Service. Embassies's minimality precludes it from reasoning about the Same-Origin Policy's content-based network restrictions; instead, Embassies addresses the underlying threats with CSN (§3.1.4). The consequence is developer freedom in network communication, but malefactors may abuse it to mount a denial-of-service (DoS) attack or a spam campaign. Today's web already allows such botnet-like attacks [34]; for example, to DoS a web server, the malefactor need only include a file (e.g., an image or JavaScript) in a popular website. Nonetheless, Embassies further enables such attacks. One mitigation would be for the client kernel to include a basic pushback mechanism [3] to allow remote hosts to squelch outbound traffic to the victim.

8 Discussion

Indexing and Mashups. Because the current web's CEI is so high-level, a vendor can easily create an app that interacts with other apps without their deliberate participation. A prominent example is web indexing, which works because the "internals" of most web content is in HTML. While Embassies permits vendors to use proprietary or obfuscated software, such behavior already occurs (e.g., Gmail's JavaScript code); baking HTML into the CEI does not guarantee hackability. In Embassies, HTML isn't required, but as with any popularity distribution, most apps will use one of a few popular DPI frameworks, and hence will allow third-party inspection. Because indexing is now so valuable, all popular DPI stacks will likely export an explicit indexing interface.

The "mashup" captures a broader category of serendipitous innovative reuse, such as data streams displayed on a map. Again, mashups interpose on the unobscured client-server traffic of ancestor applications; since those apps are likely to use popular frameworks, the same possibilities will be open. Ecosystem diversity is not enough to foil opportunistic extension; intentional obfuscation is required, a hurdle no less present in HTML than in Embassies.

Ad Blockers. Today, users can install browser extensions that interpose on apps. In Embassies, cooperating vendors could speak a bilateral protocol to a repository of extensions, but some extensions, like ad blockers, represent an adversarial relationship between user and vendor. Every user wants it, but no vendor does.

Since our CEI gives full control of an app to its the vendor, it confounds users who want to alter it in an unintended fashion. This tradeoff is deep. The client system cannot distinguish between an enhancement and a Trojan. Allowing extensions requires asking users to make that distinction, a responsibility few users can exercise correctly. We consider it worth giving up the ad blocker in exchange for a web where clicking links is always safe. Although this philosophy is new for the web, proprietary platforms such as the iPhone and Windows Phone deny unilateral app modifications.

Accessibility. Responsibility to provide accessibility falls to the vendor of each app, just as all aspects of app behavior do. However, we expect many vendors to write their applications against a higher-level DPI. Any mature DPI already incorporates accessibility features; thus any app built on such a DPI will be accessible.

Cross-Architecture Compatibility. Since our CEI specifies native-code execution, it does not solve the architecture portability problem in the CEI. We argue that architecture portability is a problem that can—and should—be solved in the vendor's software stack. One solution is to use a managed language (Java or .NET) or a portable representation (LLVM [37]) as a DPI.

DPIs based on unmanaged languages such as C or C++ can emit binaries for multiple architectures, as Linux distributions routinely do; this requires access to library source code (or recompiled libraries) as well. App vendors then face only the minor burden of hosting multiple binaries, a task easily automated, and less burdensome than dealing with today's browser incompatibilities.

On the rare occasion when a hardware company deploys a new Instruction Set Architecture (ISA), that ISA defines a new instance of the CEI. Until app vendors produce native binaries for the new ISA, the ISA company can implement, in their client kernel, an emulator for a popular ISA, as Apple did when it migrated its product line from 68K to PPC and again from PPC to x86.

GPUs. Today's web exploits the GPU by baking in further complexity, e.g., OpenGL or DirectX. Embassies' long-term solution is to treat the GPU as a CPU [7, 13, 21]. In the medium term, most deployed GPUs use segmented memory architectures adequate to isolate shader programs at GPU-load time without the client kernel understanding shader semantics. At present, even the CPU alone is pretty satisfying: Marble's CPU-rendered spinning globe (Fig. 5) is impressive.

Peripherals. Classic browsers expose printers and GPS. Does extending Embassies to include local devices erode the idea of the pico-datacenter? We think not.

Consider printing: Today, users can send photos from the Flickr app to the Snapfish app; Snapfish is a web service that includes a (remote) printer. Google Cloud Print extends the same semantics to a nearby printer. Indeed, many standalone printers already have IP interfaces. We can treat printers not as PC peripherals, but as applications that have a physical presence.

The same principle applies to other peripherals. A GPS with an IP interface need not be a PC peripheral; it may as well be an app like any other, one that gives the user control over which vendors see it. Of course, no IP hardware is required; the GPS can use a picoprocess on the client to host its IP stack.

Local storage is even simpler. Section 3.1.3 describes how apps employ a local untrusted storage service to securely store MACs and cookies. We have only implemented a RAM-based untrusted local store and a cloud-storage VFS module so far, but a disk could easily be exposed: Just a single vendor can manage the printer, Seagate might own the disk and offer untrusted, low-reliability storage, perhaps without even a UI.

Of course, we have described Embassies as a browser replacement, implying an underlying host OS; how does it interact with the host file system? Ultimately, we envision rich Embassies apps as a viable alternative to desktop OS apps. In the meantime, we envision exposing the host file system as another service, just as Google Cloud Print exposes the host printer as a service.

Deployment. Deploying a new web architecture is hard. However, Embassies apps can facilitate incremental deployment by providing a fallback for "legacy" HTTP links. With reference to Section 4.1, if caller.net's web runtime cannot resolve the name `target.org` using the PKI, it obtains and launches a web runtime which target.org might specify in a `browser.txt` file, or the caller app may supply a default.

This web runtime fetches and renders target.org's content via standard HTTP and HTML. However, the web runtime does not have a certificate chain for the label "target.org". Instead, the web runtime passes its own label (e.g., "mozilla.org") to `verify_label`. Thus, client kernel strongly authenticates the web runtime, which then attests, e.g., via its own intra-window decoration, that it is rendering content from target.org.

9 Conclusion

We propose to radically refactor the web interface to turn the client into a pico-datacenter in which app vendors run rich applications that are strongly isolated from each other. We described and implemented Embassies, a concrete, minimal CEI to support this vision, and we rebuilt existing browser-based app interactions atop the CEI. Our implementation and evaluation indicate that the CEI offers a significantly reduced TCB, yet supports a diverse set of DPIs. App and protocol performance is comparable to the existing web; app start time and memory usage is still higher than we would like, but there are clear paths towards improving them. Once native DPIs are available, and conventional apps can run in a web-like deployment, the Embassies architecture may become a compelling model for desktops or mobile platforms.

Acknowledgements

The authors gratefully thank Arun Seehra, Srinath Setty, and Xi Xiong, the interns who contributed to Embassies; Jay Lorch, James Mickens, and Mike Walfish for extensive discussions and feedback, with special thanks to James for suggesting the term "pico-datacenter"; Adrian Bonar for discussions and development work; and Alex Moshchuk, Will Scott, the anonymous reviewers, and our shepherd, Michael Piatek, for their helpful comments on the paper.

References

[1] Embassies source code. http://research.microsoft.com/embassies/, Feb. 2013.

[2] Abbott, R. P., Chin, J. S., Donnelley, J. E., Konigsford, W. L., Tokubo, S., and Webb, D. A. Security analysis and enhancements of computer operating systems. Tech. rep., Institute for Computer Sciences and Technology, National Bureau of Standards, US Department of Commerce, Apr. 1976.

[3] Andersen, D. G., Balakrishnan, H., Feamster, N., Koponen, T., Moon, D., and Shenker, S. Accountable Internet Protocol (AIP). In *Proceedings of ACM SIGCOMM* (2008).

[4] Baron, L. D. Preventing attacks on a user's history through CSS :visited selectors. http://dbaron.org/mozilla/visited-privacy, 2010.

[5] Barth, A. HTTP state management mechanism. RFC 6265 (Proposed Standard), Apr. 2011.

[6] Barth, A., Jackson, C., and Mitchell, J. C. Robust defenses for cross-site request forgery. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)* (2008).

[7] Buck, I., Foley, T., Horn, D., Sugerman, J., Fatahalian, K., Houston, M., and Hanrahan, P. Brook for GPUs: Stream computing on graphics hardware. In *SIGGRAPH* (2004).

[8] Collberg, C., Hartman, J. H., Babu, S., and Udupa, S. K. SLINKY: Static linking reloaded. In *Proceedings of the USENIX Annual Technical Conference (ATC)* (Apr. 2005).

[9] Cox, R. S., Gribble, S. D., Levy, H. M., and Hansen, J. G. A safety-oriented platform for Web applications. In *IEEE Symp. on Security & Privacy* (2006).

[10] Dhamija, R., and Tygar, J. D. The battle against phishing: Dynamic security skins. In *ACM Symposium on Usable Security and Privacy (SOUPS '05)* (July 2005).

[11] Douceur, J. R., Howell, J., Parno, B., Walfish, M., and Xiong, X. The web interface should be radically refactored. In *Proceedings of the ACM Workshop on Hot Topics in Networks (HotNets)* (2011).

[12] Esfahbod, B. *Preload: An adaptive prefetching daemon*. PhD thesis, 2006.

[13] Fatahalian, K., and Houston, M. A closer look at GPUs. *Communications of the ACM 51*, 10 (Oct. 2008).

[14] Feske, N. Introducing Genode. Talk at the Free and Open Source Software Developers' European Meeting. Slide available at http://genode.org, Feb. 2012.

[15] Feske, N., and Helmuth, C. A Nitpicker's guide to a minimal-complexity secure GUI. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)* (2005).

[16] Feske, N., and Helmuth, C. A nitpicker's guide to a minimal-complexity secure GUI. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)* (2005).

[17] Feske, N., and Helmuth, C. Design of the Bastei OS architecture. Tech. Rep. TUD-FI06-07, TU Dresden, Dec. 2006.

[18] Google. Google compute engine. http://cloud.google.com/compute/, 2012.

[19] Gosling, J., Joy, B., and Steele, G. *Java™ Language Specification*. Addison-Wesley, 1996.

[20] Grier, C., Tang, S., and King, S. T. Secure web browsing with the OP web browser. In *Proceedings of the IEEE Symposium on Security and Privacy* (2008).

[21] Gummaraju, J., Morichetti, L., Houston, M., Sander, B., Gaster, B. R., and Zheng, B. Twin peaks: A software platform for heterogeneous computing on general-purpose and graphics processors. In *Proceedings of the Conference on Parallel Architectures and Compilation Techniques (PACT)* (2010).

[22] Gupta, D., Lee, S., Vrable, M., Savage, S., Snoeren, A. C., Varghese, G., Voelker, G. M., and Vahdat, A. Difference engine: Harnessing memory redundancy in virtual machines. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (Dec. 2008).

[23] Hardy, N. The Confused Deputy (or why capabilities might have been invented). *Operating Systems Review 22*, 4 (1988).

[24] Härtig, H., Hohmuth, M., Liedtke, J., Wolter, J., and Schönberg, S. The performance of μ-kernel-based systems. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)* (1997).

[25] HOWELL, J., DOUCEUR, J. R., ELSON, J., AND LORCH, J. R. Leveraging legacy code to deploy desktop applications on the web. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (2008).

[26] HOWELL, J., JACKSON, C., WANG, H. J., AND FAN, X. MashupOS: Operating system abstractions for client mashups. In *Proceedings of the USENIX Workshop on Hot Topics in Operating Systems (HotOS)* (May 2007).

[27] JACKSON, C., AND BARTH, A. Beware of finer-grained origins. In *Proceedings of the IEEE Web 2.0 Security and Privacy Workshop (W2SP)* (2008).

[28] JANG, D., VENKATARAMAN, A., SAWKA, G. M., AND SHACHAM, H. Analyzing the crossdomain policies of Flash applications. In *Proceedings of the IEEE Web 2.0 Security and Privacy Workshop (W2SP)* (2011).

[29] JOHNS, M. *Code Injection Vulnerabilities in Web Applications Exemplified at Cross-Site Scripting*. PhD thesis, University of Passau, 2009.

[30] KAASHOEK, M. F., ENGLER, D. R., GANGER, G. R., NO, H. M. B., HUNT, R., MAZIÈRES, D., PINCKNEY, T., GRIMM, R., JANNOTTI, J., AND MACKENZIE, K. Application performance and flexibility on Exokernel systems. In *SOSP* (1997).

[31] KELLEY, P., CONSOLVO, S., CRANOR, L., JUNG, J., SADEH, N., AND WETHERALL, D. An conundrum of permissions: Installing applications on an android smartphone. In *Workshop on Usable Security* (2012).

[32] Kernel-based virtual machine. http://www.linux-kvm.org. Accessed May, 2012.

[33] KROVETZ, T., AND DAI, W. VMAC: Message authentication code using universal hashing. Internet Draft: http://http://fastcrypto.org/vmac/draft-krovetz-vmac-01.txt, Apr. 2007.

[34] LAM, V. T., ANTONATOS, S., AKRITIDIS, P., AND ANAGNOSTAKIS, K. G. Puppetnets: Misusing web browsers as a distributed attack infrastructure. In *Proceedings of the ACM Conference on Computer and Communications Security* (2006).

[35] LAMPSON, B., ABADI, M., BURROWS, M., AND WOBBER, E. Authentication in distributed systems: Theory and practice. *ACM Transactions on Computer Systems 10*, 4 (Nov. 1992), 265–310.

[36] LANGLEY, A. Chromium's seccomp sandbox. Blog post, 2009. http://www.imperialviolet.org/2009/08/26/seccomp.html.

[37] LATTNER, C. LLVM: An infrastructure for multi-stage optimization. Master's thesis, UIUC, 2002.

[38] LEVIN, D., DOUCEUR, J. R., LORCH, J. R., AND MOSCIBRODA, T. TrInc: Small trusted hardware for large distributed systems. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (2009).

[39] MAHAJAN, R., PADHYE, J., RAGHAVENDRA, R., AND ZILL, B. Eat all you can in an all-you-can-eat buffet: A case for aggressive resource usage. In *Proceedings of the ACM Workshop on Hot Topics in Networks (HotNets)* (2008).

[40] MERKLE, R. C. A certified digital signature. In *Proceedings of CRYPTO* (1989), pp. 218–238.

[41] MICKENS, J., AND DHAWAN, M. Atlantis: Robust, extensible execution environments for Web applications. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)* (2011).

[42] MICROSOFT MSDN. How to set network capabilities (Windows). http://msdn.microsoft.com/en-us/library/windows/apps/hh770532.aspx, Dec. 2012.

[43] MICROSOFT MSDN. Virtual machines. http://msdn.microsoft.com/en-us/library/windowsazure/jj156003.aspx, June 2012.

[44] MOSHCHUK, A., AND WANG, H. J. Resource management for web applications in ServiceOS. Tech. Rep. MSR-TR-2010-56, Microsoft Research, May 2010.

[45] MOSHCHUK, A., WANG, H. J., AND LIU, Y. Content-based isolation: Rethinking isolation policy in modern client systems. Tech. Rep. MSR-TR-2012-82, Microsoft Research, Aug. 2012.

[46] NILS. Building Android sandcastles in Android's sandbox. Black Hat, https://media.blackhat.com/bh-ad-10/Nils/Black-Hat-AD-2010-android-sandcastle-wp.pdf, Oct. 2010.

[47] O'NEILL, S. 'Consumerization of IT' taking its toll on IT managers. *CIO* (Sept. 2011).

[48] Parno, B., Lorch, J. R., Douceur, J. R., Mickens, J., and McCune, J. M. Memoir: Practical state continuity for protected modules. In *Proceedings of the IEEE Symposium on Security and Privacy* (May 2011).

[49] Porter, D. E., Boyd-Wickizer, S., Howell, J., Olinsky, R., and Hunt, G. C. Rethinking the Library OS from the Top Down. In *Proceedings of the ACM Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2011).

[50] Roelofs, G. libpng. Software distribution. http://www.libpng.org/pub/png/libpng.html.

[51] Roesner, F., Kohno, T., Moshchuk, A., Parno, B., Wang, H. J., and Cowan, C. User-driven access control: Rethinking permission granting in modern operating systems. In *Proceedings of the IEEE Symposium on Security and Privacy* (May 2012).

[52] Schechter, S. E., Dhamija, R., Ozment, A., and Fischer, I. The emperor's new security indicators. In *Proceedings of the IEEE Symposium on Security and Privacy* (2007), pp. 51–65.

[53] Shapiro, J. S., Vanderburgh, J., Northup, E., and Chizmadia, D. Design of the EROS trusted window system. In *Proceedings of the USENIX Security Symposium* (2004).

[54] Sirer, E. G., de Bruijn, W., Reynolds, P., Shieh, A., Walsh, K., Williams, D., and Schneider, F. B. Logical attestation: An authorization architecture for trustworthy computing. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)* (2011).

[55] Tang, S., Mai, H., and King, S. T. Trust and Protection in the Illinois Browser Operating System. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (2010).

[56] Twotoasts. Midori. http://twotoasts.de/index.php/midori/, 2012.

[57] Wang, H. J., Fan, X., Jackson, C., and Howell, J. Protection and communication abstractions for web browsers in MashupOS. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)* (Oct. 2007).

[58] Wang, H. J., Grier, C., Moshchuk, A., King, S. T., Choudhury, P., and Venter, H. The multi-principal OS construction of the Gazelle web browser. In *USENIX Security Symposium* (2009).

[59] Wang, H. J., Moshchuk, A., and Bush, A. Convergence of desktop and web applications on a multi-service OS. In *USENIX HotSec Workshop* (2009).

[60] Wang, R., Chen, S., and Wang, X. Signing me onto your accounts through Facebook and Google: a traffic-guided security study of commercially deployed single-sign-on web services. In *Proceedings of the IEEE Symposium on Security and Privacy* (May 2012).

[61] WebKit. SunSpider JavaScript Benchmark. Version 0.9.1 at http://www.webkit.org/perf/sunspider/sunspider.html, 2012.

[62] The WebKit open source project. http://www.webkit.org/, 2012.

[63] Weinberg, Z., Chen, E. Y., Jayaraman, P. R., and Jackson, C. I still know what you visited last summer: User interaction and side-channel attacks on browsing history. In *Proceedings of the IEEE Symposium on Security and Privacy* (2011).

[64] Wheeler, D. A. SLOCCount. Software distribution. http://www.dwheeler.com/sloccount/.

[65] Ye, E., and Smith, S. Trusted paths for browsers. In *Proceedings of the 11th USENIX Security Symposium* (Aug. 2002).

[66] Yee, B., Sehr, D., Dardyk, G., Chen, J. B., Muth, R., Ormandy, T., Okasaka, S., Narula, N., and Fullagar, N. Native client: A sandbox for portable, untrusted x86 native code. In *Proceedings of the IEEE Symposium on Security & Privacy* (2009).

[67] Zalewski, M. Browser security handbook: Same-origin policy. Online handbook. http://code.google.com/p/browsersec/wiki/Part2.