Gibraltar: Exposing Hardware Devices to Web Pages Using AJAX

Kaisen Lin
UC San Diego

David Chu James Mickens
Li Zhuang Feng Zhao
Microsoft Research

Jian Qiu
National University of Singapore

Abstract

Gibraltar is a new framework for exposing hardware devices to web pages. Gibraltar's fundamental insight is that JavaScript's AJAX facility can be used as a hardware access protocol. Instead of relying on the browser to mediate device interactions, Gibraltar sandboxes the browser and uses a small device server to handle hardware requests. The server uses native code to interact with devices, and it exports a standard web server interface on the localhost. To access hardware, web pages send device commands to the server using HTTP requests; the server returns hardware data via HTTP responses.

Using a client-side JavaScript library, we build a simple yet powerful device API atop this HTTP transfer protocol. The API is particularly useful to developers of mobile web pages, since mobile platforms like cell phones have an increasingly wide array of sensors that, prior to Gibraltar, were only accessible via native code plugins or the limited, inconsistent APIs provided by HTML5. Our implementation of Gibraltar on Android shows that Gibraltar provides stronger security guarantees than HTML5; furthermore, it shows that HTTP is responsive enough to support interactive web pages that perform frequent hardware accesses. Gibraltar also supports an HTML5 compatibility layer that implements the HTML5 interface but provides Gibraltar's stronger security.

1. Introduction

Web browsers provide an increasingly rich execution platform. Unfortunately, browsers have been slow to expose low-level hardware devices to JavaScript [8], the most popular client-side scripting language. This limitation has become particularly acute as sensor-rich devices like phones and tablets have exploded in popularity. A huge marketplace has arisen for mobile applications that leverage data from accelerometers, microphones, GPS units, and other sensors. Phones also have increasingly powerful computational and storage devices. For example, graphics processors (GPUs) are already prevalent on phones, and using removable storage devices like SD cards, modern phones can access up to 64GB of persistent data.

Because JavaScript has traditionally lacked access to such hardware, web developers who wanted to write device-aware applications were faced with two unpleasant choices: learn a new plugin technology like Flash which is not supported by all browsers, or learn a platform's native application language (e.g, the Win32 API for Windows machines, or Java for Android). Both choices limit the portability of the resulting applications. Furthermore, moving to native code eliminates a key benefit of the web delivery model—applications need not be installed, but merely navigated to.

1.1 A Partial Solution

To remedy these problems, the new HTML5 specification [10] introduces several ways for JavaScript to access hardware. At a high-level, the interfaces expose devices as special objects embedded in the JavaScript runtime. For example, the `<input>` tag [24] can reflect a web cam object into a page's JavaScript namespace; the page reads or writes hardware data by manipulating the properties of the object. Similarly, HTML5 exposes geolocation data through the `navigator.geolocation` object [27]. Browsers implement the object by accessing GPS devices, or network cards that triangulate signals from wireless access points.

Given all of this, there are two distinct models for creating device-aware web pages:

- Applications can be written using native code or plugins, and gain the performance that results from running close to the bare metal. However, users must explicitly install the applications, and the applications can only run on platforms that support their native execution environment.
- Alternatively, applications can be written using cross-platform HTML5 and JavaScript. Such applications do not require explicit installation, since users just navigate to the application's URL using their browser. However, as shown in the example above, HTML5 uses an inconsistent set of APIs to name and query each device, making it difficult to write generic code. Furthermore, by exposing devices through extensions of the JavaScript interpreter, the entire JavaScript runtime becomes a threat surface for a malicious web page trying to access unauthorized hardware—once a web page has compromised the browser, nothing stands between it and the user's devices. Unfortunately, modern browsers are large, complex, and have many exploitable vulnerabilities [4, 30, 34]. On mobile devices, browsers represent a key infection vector for malicious pages that steal SMS information [21], SD card data [23], and other private user information.

Ideally, we want the best of both worlds—device-aware, cross-platform web pages that require no installation, but whose security does not depend on a huge trusted computing base like a browser.

1.2 Our Solution: Gibraltar

Our new system, called Gibraltar, uses HTTP as a hardware access protocol. Web pages access devices by issuing AJAX requests to a *device server*, a simple native code application which runs in a separate process on the local machine and exports a web server interface on the localhost domain. If a hardware request is authorized, the device server performs the specified operation and returns any data using a standard HTTP response. Users authorize individual web domains to access each hardware device, and the device server authenticates each AJAX request by ensuring that the referrer field [7] represents an authorized domain.

Unlike HTML5, Gibraltar does not require the browser to be fully trusted. Indeed, in Gibraltar, the browser is sandboxed and incapable of accessing most devices. However, a corrupted or malicious browser can send AJAX requests to the device server which contain snooped referrer fields from authorized user requests. To limit these attacks, Gibraltar uses *capability tokens* and *sensor widgets* [12]. Before a web page can access hardware, it must fetch a token from the device server. The page must tag subsequent hardware requests with the fresh capability.

To prevent a malicious browser from surreptitiously requesting capabilities from the device server, Gibraltar employs sensor widgets. Sensor widgets are ambient GUI elements like system tray icons that indicate which hardware devices are currently in use, and which web pages are using them. Sensor widgets help a user to detect discrepancies between the set of devices that she expects to be in use, and the set of devices that are actually in use. Thus, sensor widgets allow a user to detect when a compromised browser is issuing hardware requests that the user did not initiate.

Using these mechanisms, a compromised browser in Gibraltar has limited abilities to independently access hardware (§5). However, a malicious browser is still the conduit for HTTP traffic, so it can snoop on data that the user has legitimately fetched and send that data to remote hosts. Gibraltar does not stop these kinds of attacks. However, Gibraltar is complementary to information flow systems like TightLip [39] that can prevent such leaks.

1.3 Advantages of Gibraltar

Gibraltar's device protocol has four primary advantages:

- **Ease of Deployment:** Gibraltar allows device-aware programs to be shipped as web applications that do not need to be installed. The device server does need to be installed, but it can ship alongside the browser and be installed at the same time that the browser itself is installed.
- **Security:** Compared to HTML5-style approaches which expose hardware by extending the JavaScript interpreter, Gibraltar has a much smaller attack surface. Gibraltar's HTTP protocol is a narrow waist for hardware accesses, and the device server is much simpler than a full-blown web browser; for example, our device server for Android phones is only 7613 lines of strongly typed Java code, instead of the million-plus lines of C++ code found in popular web browsers. Using capability tokens and sensor widgets, Gibraltar can also prevent (or at least detect) many attacks from malicious web pages and browsers. HTML5 cannot stop or detect any of these attacks.
- **Usability:** An HTTP device protocol provides a uniform naming scheme for disparate devices, and makes it easy for pages to access non-local devices. For example, a page running on a user's desktop machine may want to interact with sensors on the user's mobile phone. If a Gibraltar device server runs on the phone, the page can access the remote hardware using the same interface that it uses for local hardware—the only difference is that the device server is no longer in the localhost domain.
- **Backwards Compatibility:** It is straightforward to map HTML5 device commands to Gibraltar calls. Thus, to run a preexisting HTML5 application atop Gibraltar, a developer can simply include a translation library that converts HTML5 calls to Gibraltar calls but preserves Gibraltar's security advantages. The library can use Mugshot-style interpositioning [19] to intercept the HTML5 calls.

Since Gibraltar uses HTTP to transport hardware data, a key question is whether this channel has sufficient bandwidth and responsiveness to support real device-driven applications. To answer this question, we wrote a device server for Android mobile phones, and modified four non-trivial applications to use the Gibraltar API. Our evaluation shows that Gibraltar is fast enough to support real-time programs like games that require efficient access to hardware data.

2. Design

Gibraltar uses privilege separation [29] to provide a web page with hardware access. The web page, and the enclosing browser which executes the page's code, are both untrusted. Gibraltar places the browser in a sandbox which prevents direct access to Gibraltar-mediated devices. The small, native code device server resides in a separate process from the browser, and executes hardware requests on behalf of the page, exchanging data with the page via HTTP.

As shown in Figure 1, a Gibraltar-enabled page includes a JavaScript file called `hardware.js`. This library implements the public Gibraltar API. `hardware.js` fetches authentication tokens as described in Section 2.1, and translates page-initiated hardware requests into AJAX fetches as described in Section 3. `hardware.js` also receives and deserializes the responses. Note that `hardware.js` is merely a convenience library that makes it easier to program against Gibraltar's raw AJAX protocol; Gibraltar does not trust `hardware.js`, and it does not rely on `hardware.js` to enforce the security properties described in Section 5.

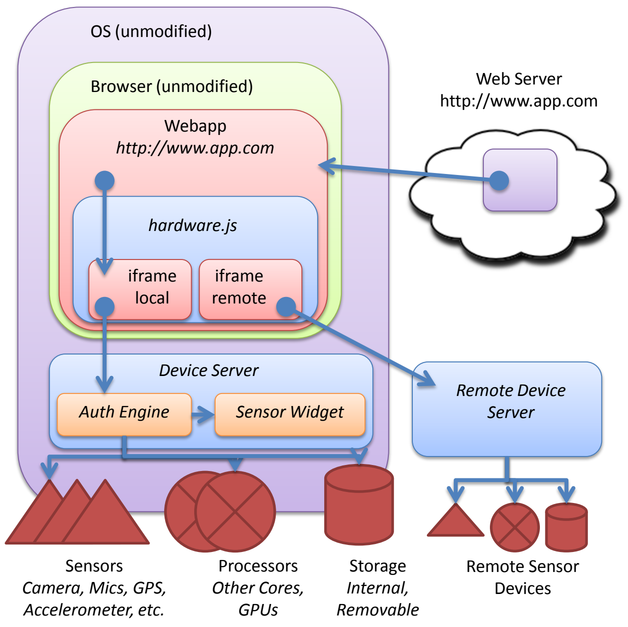


Figure 1. Gibraltar Architecture.

```
void handle_request(req){
   resp = new AJAXResponse();
   switch(req.type){
     case OPEN_SESSION:
        if(!active_tokens.contains(req.referrer)){
           resp.result = "TOKEN:" + makeNewToken();
           active_tokens[req.referrer] = resp.result;
        }
        break;

     case DEVICE_CMD:
        if(!authorized_domains[req.device].contains(
                                        req.referrer) ||
           (active_tokens[req.referrer] == null) ||
           (active_tokens[req.referrer] != req.token)){
           resp.result = "ACCESS DENIED";
        }else{
           resp.result = access_hardware(req.device,
                                          req.cmd);
           sensor_widgets.alert(req.referrer,
                                req.device);
        }
        break;

     case CLOSE_SESSION:
        if(active_tokens[req.referrer] == req.token)
            active_tokens.delete(req.referrer);
        break;
   }
   sendResponse(resp);
}
```

Figure 2. Pseudocode for device server.

Note that the device server resides in the localhost domain, whereas the Gibraltar-enabled page emanates from a different, external origin. By default, the same-origin policy would prevent the `hardware.js` in the web page from fetching cross-origin data from the localhost server. However, using the `Access-Control-Allow-Origin` HTTP header [37], the device server can instruct the browser to allow the cross-origin Gibraltar fetches. This header is supported by modern browsers like IE9 and Firefox 4+. In older browsers, `hardware.js` communicates with the device server using an invisible frame with a localhost origin; this frame exchanges Gibraltar data with the regular application frame using `postMessage()`. Similarly, Gibraltar can use a remote-origin frame to deal with off-platform devices.

2.1 Authenticating Hardware Requests

In Gibraltar, device management consists of three tasks: manifest authorization, session establishment, and session teardown. Figure 2 provides the relevant pseudocode in the device server. We discuss this code in more detail below.

Manifest authorization: On mobile devices like Android, users authorize individual applications to access specific hardware devices. Similarly, in Gibraltar, users authorize individual web domains like `cnn.com` to access individual hardware devices. When a page contacts the device server for the first time, the page includes a *device manifest* in its HTTP request. The manifest is simply a list of devices that the page wishes to access. The device server presents this manifest to the user and asks whether she wishes to grant the specified access permissions to the page's domain. If so, the device server stores these permissions in a database. Subsequent page requests for devices in the manifest will not require explicit user action, but if the page requests access to a new device, the user must approve the new permission.

Session management: Since Gibraltar hardware requests are expressed as HTTP fetches, a natural way for the device server to authenticate a request is to inspect its referrer field [7]. This is a standard HTTP field which indicates the URL (and thus the domain) of the page which generated the request. Unfortunately, a misbehaving browser can subvert this authentication scheme by examining which domains successfully receive hardware data, and then generating fake requests containing these snooped referrer fields. This is essentially a replay attack on a weak authenticator.

To prevent these replay attacks, the device server grants a *capability token* to each authorized web domain. Before a page in domain `trusted.com` can access hardware, it must send a session establishment message to the device server. The device server examines the referrer of the HTTP message and checks whether the domain has already been granted a token. If not,[1] the server generates a unique token, stores the mapping between the domain and that token, and sends the token to the page. Later, when the page sends an actual hardware request, it includes the capability token in its AJAX message. If the token does not match the mapping found in the device server's table, the server ignores the hardware request.

[1] We restrict each domain to a single token for security reasons that we describe in Section 5.1. However, this restriction does not prevent a domain from opening multiple device-aware web pages on a client—the pages can inform each other of the domain's token using the JavaScript `postMessage()` API.

A page sends a session teardown message to the device server when it no longer needs to access hardware, e.g., because the user wants to navigate to a different page. Upon receipt of the teardown message, the server deletes the relevant domain/token mapping. `hardware.js` can detect when a page is about to unload by registering a handler for the JavaScript `unload` event.

Sensor widgets: Given this capability scheme, a misbehaving browser that can only spoof referrers cannot fraudulently access hardware—the browser must also steal another domain's token, or retrieve a new one from the device server. As we discuss in Section 5, cross-domain token stealing is difficult if the browser uses memory isolation to partition domains. However, nothing prevents a browser from autonomously downloading a new security token in the background under the guise of an authorized domain, and then using this token in its AJAX requests. To prevent this attack, we use sensor widgets [12], which are ambient GUI elements like system tray icons that glow, make a noise, or otherwise indicate when a particular hardware device is in use. Sensor widgets also indicate the domains which are currently accessing hardware. Thus, if the browser tries to autonomously access hardware using a valid token, the activity will trigger the sensor widgets, alerting the user to a hardware request that she did not initiate.

The sensor widgets are implemented within the device server, not the browser. However, the browser can try to elude the widgets in several ways. In Section 5, we provide a fuller analysis of Gibraltar's security properties.

2.2 The Gibraltar API

Figure 3 lists the client-side Gibraltar API. Before a web page can issue hardware commands, it must get a new capability token via `createSession()`. Then, it must send its device manifest to the device server via `requestAccess()`. The device server presents the manifest to the user and asks her to validate the requested hardware permissions.

2.2.1 Sensor API

To provide access to sensors like cameras, accelerometers, and GPS units, Gibraltar provides a one-shot query interface and a continuous query interface. In keeping with JavaScript's event-driven programming model, `singleQuery()` and `continuousQuery()` accept an application-defined callback which Gibraltar invokes when the hardware data has arrived. The functions also accept the name of the device to query, and a device-specific `params` value which controls sensor-specific properties like the audio sampling bitrate. `continuousQuery()` takes an additional parameter representing the query frequency.

Different devices will define different formats for the `params` object, and different formats for the returned device data. However, much like USB devices, Gibraltar devices fall into a small set of well-defined classes such as storage devices, audio devices, and video devices. Thus, web pages can program against generic Gibraltar interfaces to each class; the device server and `hardware.js` can encapsulate any device-specific eccentricities.

Figure 3 also describes a sensor management interface. The power controls allow a page to shut off devices that it does not need; the device server ensures that a device is left on if at least one application still needs it. `sensorAdded()` and `sensorRemoved()` let applications register callbacks which Gibraltar fires when devices arrive or leave. These events are useful for off-platform devices like Bluetooth headsets and Nike+ shoe sensors [22].

2.2.2 Processor API

Multi-core processors and programmable GPUs are already available on desktops, and they are starting to ship on mobile devices. To let web pages access these extra cores, Gibraltar exports a simple multi-processor computing model inspired by OpenCL [13], a new specification for programming heterogeneous processors.

A Gibraltar *kernel* represents a computational task to run on a core. Kernels are restricted to executing two types of predefined functions. *Primitive functions* are geometric, trigonometric, or comparator operations. Gibraltar's primitive functions are similar to those of OpenCL. *Built-in functions* are higher-level functions that we have identified as particularly useful for processing hardware data. Examples of such functions are FFT transforms and matrix operations.

A web page passes a kernel to Gibraltar by calling `enqueueKernel()`. To execute a parallel vector computation with that kernel, the page calls `setKernelData()` with a vector of arguments; Gibraltar will instantiate a new copy of the kernel for each argument and run the kernels in parallel. A web page can also create a computation pipeline by calling `enqueueKernel()` multiple times with the same or different kernel. Gibraltar will chain the kernels' inputs and outputs in the order that the kernels were passed to `enqueueKernel()`. The page sets the input data for the pipeline by passing a scalar value to `setKernelData()`.

Once an application has configured its kernels, it calls `executeKernels()` to start the computation. Gibraltar distributes the kernels to the various cores in the system, coordinates cross-kernel communication, and fires an application-provided callback when the computation finishes.

2.2.3 Storage API

The final set of calls in Figure 3 provide a key/value storage interface. The namespace is partitioned by web domain and by storage device; a web domain can only access data that resides in its partitions. To support removable storage devices, Gibraltar fires connection and disconnection events like it does for off-platform sensors like Bluetooth headsets.

HTML5 DOM storage [11] also provides a key-value store. However, DOM storage is limited to non-removable

Call	Description
`createSession()`	Get a capability token from the device server.
`destroySession()`	Relinquish a capability token.
`requestAccess(manifest)`	Ask for permission to access certain devices.
`singleQuery(name, params)`	Get a single sensor sample.
`continuousQuery(name, params, period)`	Start periodic fetch of sensor samples.
`startSensor(name)`	Turn on a sensor.
`stopSensor(name)`	Turn off a sensor.
`sensorAdded(name)`	Upcall fired when a sensor is added.
`sensorRemoved(name)`	Upcall fired when a sensor is removed.
`getSensorList()`	Get available sensors.
`enqueueKernel(kernel)`	Queue a computation kernel for execution.
`setKernelData(parameters)`	Set the input data for the computation pipeline.
`executeKernels()`	Run the queued kernels on the input data.
`put(storename,key,value)`	Put value by key.
`get(storename,key)`	Get value by key.

Figure 3. Summary of `hardware.js` API. All calls implicitly require a security token and callback function.

media, and it does not explicitly expose the individual devices which are used for the underlying stable storage.

2.3 Remote device access

As we mentioned earlier, some devices may reside off-platform. If those devices run a Gibraltar server which accepts external connections, a web page can seamlessly access those devices using the same interface that it uses for local ones. This capability enables many interesting applications. For example, in Section 6, we evaluate a game that runs on a desktop machine but uses a mobile phone with an accelerometer as a motion-sensitive controller. In this example, the web page runs on the desktop machine, but the device server runs on the phone.

A device server accepts connections from localhost clients by default (subject to the authentication rules described in Section 2.1). For security reasons, a device server should reject connections from arbitrary remote clients. Thus, users must explicitly whitelist each external IP address or dynamic DNS name [35] that wishes to communicate with a device server. This is accomplished in a fashion similar to how the user authorizes device manifests (§2.1).

2.4 Sandboxing the Browser

Gibraltar is agnostic about the mechanism that prevents the browser from accessing Gibraltar devices. For example, mobile platforms like Android, iOS, and the Windows Phone provide device ACLs that makes it easy to prohibit applications from accessing forbidden hardware. Gibraltar is also compatible with other isolation techniques like hardware virtualization or binary rewriting.

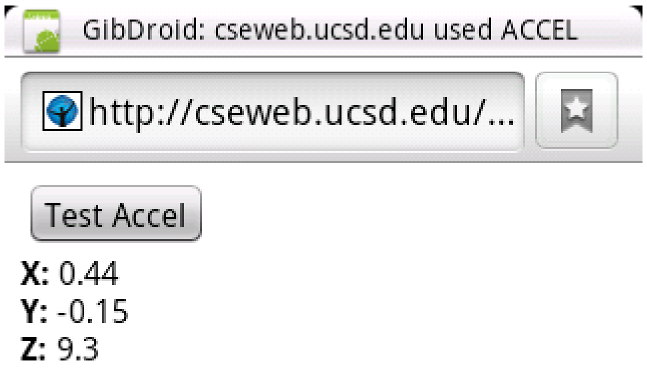
3. Implementation

Client-side Library: `hardware.js` encodes device requests using a simple XML string. Each request contains a security token, an action to perform, the target device, and optional device-specific parameters. For example, a request to record microphone data includes a parameter that represents the recording duration. Device responses are also encoded using XML. The response specifies whether the request succeeded, and any data associated with the operation. The device server encodes binary data in `Base64` format so that `hardware.js` can represent data as JavaScript strings.

Android Device Server: On Android 2.2, we implemented the device manager as a servlet for the i-jetty web server [2]. A servlet is a Java software module that a web server invokes to handle certain URL requests. The Gibraltar servlet handles all requests for Gibraltar device URLs. The servlet performs the authentication checks described in Section 2.1, accesses hardware using native code, and returns the serialized results. We refer to our Android implementation of Gibraltar as GibDroid.

The GibDroid device server has different probing policies for low throughput sensors and high throughput sensors. For low throughput devices like cameras, GibDroid accesses the sensor on demand. For devices like accelerometers that have a high data rate, the GibDroid server continuously pulls data into a circular buffer. When a page queries the sensor, the device server returns the entire buffer, allowing multiple data points to be fetched in a single HTTP round trip. Currently, GibDroid provides access to accelerometers, GPS units, cameras (both single pictures and streaming video), microphones, local storage, and native computation kernels.

Before a web page can receive hardware data from the device server, it must engage in a TCP handshake with the server and send an HTTP header. For devices with high data rates like accelerometers and video cameras, creating an HTTP session for each data request can hurt performance, even with sample batching. Thus, GibDroid allows the de-


GibDroid: cseweb.ucsd.edu used ACCEL

http://cseweb.ucsd.edu/...

Test Accel

X: 0.44
Y: -0.15
Z: 9.3

Figure 4. GibDroid uses the Android notification bar to hold sensor widgets.

vice server to use Comet-style [3] data pushes. In this approach, `hardware.js` establishes a persistent HTTP connection with the device server using a "forever frame." Unlike a standard frame, whose HTML size is declared in the HTTP response by the server, a forever frame has an indefinite size, and the server loads it incrementally, immediately pushing a new HTML chunk whenever new device data arrives. Each chunk is a dynamically generated piece of JavaScript code; the code contains a string variable representing the new hardware data, and a function call which invokes an application-defined handler. Forever frames are widely supported by desktop browsers, but currently unsupported by many mobile browsers. Thus, the device server reverts to request-response for mobile browsers.

GibDroid can stream accelerometer data and video frames using Comet data pushes. To handle video, our current implementation uses data URIs [18] to write `Base64`-encoded data to an `<image>` tag[2]. Many current browsers limit data URIs to 32 KB data; thus, data URIs are only appropriate for sending previews of larger video images. Our current GibDroid implementation displays video frames with a pixel resolution of 530 by 380. The device server uses Android's `setPreviewCallback()` call to access preview-quality images from the underlying video camera.

GibDroid supports the execution of kernel functions. However, our evaluation Android phone does not have secondary processing cores. Therefore, GibDroid kernels run in separate Java threads that time-slice the single processor with other applications.

As shown in Figure 4, GibDroid places sensor widgets in the standard notification bar that exists in all Android phones. The notification bar is a convenient place to put the widgets because users are already accustomed to periodically scanning this area for program updates. We are still experimenting with the visual presentation for the widgets, so Figure 4 represents a work-in-progress.

[2] In the next version of Gibraltar, `hardware.js` will write video frames to the bitmap of an HTML5 `Canvas` object [10].

Windows PC Device Server: We also wrote a device server for Windows PCs. This device server, written in C#, currently only provides access to the hard disk, but it is the target of active development. In Section 6.1, we use this device server to compare Gibraltar's performance on a multi-core machine to that of HTML5.

4. Applications

In this section, we describe four new applications which use the Gibraltar API to access hardware. We evaluate the performance of these applications in Section 6.

Our first application is a mapping tool similar to MapQuest [17]. This web page uses GPS data to determine the user's current location. It also uses Gibraltar's storage APIs to load cached maps tiles. More specifically, we assume that the phone's operating system prefetches map tiles, similar to how the Songo framework prefetches mobile ads [14]. The operating system stores the map tiles in the file system; for each cached tile, the OS adds the key/value pair `(tileId,fileSystemLocation)` to the mapping application's Gibraltar storage area. When the map application loads, it determines the user's current location and calculates the set of tiles to fetch. For each tile, it consults the tile index in Gibraltar storage to determine if the tile resides locally. If it does, the page loads the tile using an `<img>` tag with a `file://` origin; otherwise, the page uses a `http://` origin to fetch the image from the remote tile server.

The popular native phone application *Shazam!* identifies songs that are playing in the user's current environment. *Shazam!* does this by capturing microphone data and applying audio fingerprint algorithms. Inspired by *Shazam!*, we built Gibraltar Sound, a web application that captures a short sound clip and classifies it as *music*, *conversation*, *typing*, or *other ambient sound*. To classify sounds, we used Mel-frequency cepstrums (MFCC) for feature extraction, and Gaussian Mixture Models (GMM) for inference [16]. We implemented MFCC and GMM as native built-in kernels.

Our final applications leverage Gibraltar's ability to access off-platform devices. These pages load on a desktop machine's browser, but use Gibraltar to turn a mobile phone into a game controller. The first application, Gibraltar Paint, is a simple painting program in which user gestures with the phone are converted into brush strokes on a virtual canvas. Gestures are detected using the phone's accelerometer.

We also modified a JavaScript version of Pacman [15] to use a Gibraltar-enabled phone as a controller for a game loaded on the desktop browser—tilting the phone in a direction will cause Pacman to move in that direction. HTML5 cannot support the latter two applications because it lacks an API for remote device access.

5. Security

Any mechanism for providing hardware data to web pages must grapple with two questions. First, can it ensure that each device request was initiated by the user instead of a misbehaving browser? Second, once the hardware data has been delivered to browser, can the system prevent the browser from modifying or leaking that data in unauthorized ways? Gibraltar only addresses the first question, but it is complementary to systems that address the second. In Section 5.1, we describe the situations in which Gibraltar can and cannot prevent fraudulent hardware access. In Section 5.2, we describe how Gibraltar can be integrated with a taint tracking system to minimize unintended data leakage.

5.1 Authenticating Hardware Requests

In Gibraltar, there are five kinds of security principals: the user, the Gibraltar device server, the underlying operating system, web pages, and the web browser. Gibraltar does not trust the last two principals. More specifically, Gibraltar's security goal is to prevent unauthorized web pages from accessing hardware data, and faulty web browsers from autonomously fetching such data. Gibraltar assumes that the OS properly sandboxes the browser, and that the OS prevents the browser from directly accessing Gibraltar-mediated hardware; Gibraltar is agnostic to the particular sandboxing mechanism that is used, e.g., binary rewriting, virtual machines, or OS-enforced device ACLs provided by platforms like Android and iOS. Gibraltar assumes that the device server is implemented correctly, that the user can inform the device server of authorized web sites without interference, and that the operating system prevents the web browser from directly tampering with the device server. Thus, the only way that a faulty web page or browser can access hardware is by subverting the AJAX device protocol.

As shown in Figure 2, the device server will only respond to a hardware request if the request has an authorized referrer field and a valid authentication token; furthermore, the authorized domain cannot have another open session involving a different token. Thus, Gibraltar's security with respect to device D can be evaluated in the context of three parameters: whether the attacker can fake referrer fields, whether the attacker can steal tokens from domains authorized to access D, and whether the user currently has a legitimate, active frame belonging to a legitimately authorized domain. Figure 5(a) provides concrete threat examples that correspond to whether an attacker can fake referrers or steal tokens.

Figure 5(b) shows Gibraltar's attack resilience when the user does not have an authorized frame open. Figure 5(c) shows the attacker's power when the user has opened an authorized frame. In both cases, we see that an attacker cannot fraudulently access hardware if he cannot fake referrer fields. If the attacker can fake referrer fields, then his ability to fraudulently access device D depends on whether the user has already opened a frame for a domain that is authorized to access D. If no such frame is open, the attacker can successfully fetch an authentication token from the device server, since the domain will not have an outstanding token in circulation. The attacker's first hardware request will pass the device server's authentication tests, since the referrer will be authorized and the token will be valid. However, the device server will trigger the appropriate sensor widget for D, indicating the (spoofed) trusted domain that is accessing that device. At this point, the user can realize that she has not legitimately opened a frame in that domain, and she can shut down her browser or take other remediating steps. Although the browser has gained limited access to hardware data, Gibraltar can work in concert with a taint tracking system to prevent the data from being externalized (§5.2).

Now suppose that the attacker can fake referrer fields, and the user does have an authorized frame open (this is the right column of Figure 5(c)). If the browser uses a Gazelle-style architecture [36] and strongly isolates the attacker page from the authorized page, the attacker cannot inspect the token in the authorized page. Thus, the attacker must request a new token from the device server. However, the server will refuse this request because the domain in the referrer field will already have a token.

If the attacker can steal tokens *and* fake referrers, and the user already has an authorized frame open, then nothing prevents the attacker from opportunistically hiding his hardware requests within the background traffic from the legitimately authorized frame. Although current browsers do provide a modicum of domain isolation (e.g., via IE's process-per-tab model, or Chrome's process-per-site-instance model [31]), commercial browsers do not implement Gazelle-strength isolation. However, browsers are continually moving towards stronger isolation models, so we believe that soon, cross-frame token stealing will be impossible.

Gibraltar assumes that the operating system correctly routes packets to the device server. Thus, the device server can reject arbitrary connections from off-platform entities by verifying that the source in each AJAX request has a localhost IP address. If a user wants to associate a device server with a web page that resides off-platform, she must whitelist the external IP address, or notify the device server and the web page of a shared secret which enables the device server to detect trusted external clients. For example, the client web page might generate a random number and include this number in the first AJAX request that it sends to the device server. When the server receives this request, it can present the nonce to the user for verification.

Malicious local applications that are not web pages can also try to access hardware by contacting the device server. Sensor widgets provide some defense, but using tools like Linux's `lsof` or Windows' `Process Explorer`, the device server can simply reject localhost connections from programs that are not hosted within a web browser.

5.2 Securing Returned Device Data

The browser acts as the conduit for all AJAX exchanges, and it can arbitrarily inspect the JavaScript runtimes inside each page. Thus, once the browser has received hardware data (either because a user legitimately fetched it, or because the

	Cannot fake referrer	**Can fake referrer**
Cannot steal token	An unauthorized web page seeks hardware access while running in an uncompromised browser.	A malicious page subverts its container in a browser with per-frame memory isolation, but the page cannot peek into the address spaces of other frames in different processes.
Can steal token	The browser is uncompromised, but an authorized page in one tab willingly shares its token with an attacker page in another tab.	A malicious page subverts a monolithic, single-process browser, allowing it to steal a token from another authorized tab that is running.

(a) Example attacker scenarios.

	Cannot fake referrer	**Can fake referrer**
Cannot steal token	Attack prevented (attacker cannot create an authorized referrer for its hardware request).	Attack detected (attacker can spoof referrer field from trusted domain and get a new token, but sensor widgets alert user to the fact that trusted domain X is accessing hardware but the user hasn't opened a page from X).
Can steal token	Attack prevented (attacker cannot create an authorized referrer for its hardware request).	Attack detected (no legitimately authorized page exists from which the attacker can steal a token; thus, attacker is forced to download a new token as above, and is detected by the sensor widgets).

(b) Gibraltar attack resilience (no legitimately authorized page is running).

	Cannot fake referrer	**Can fake referrer**
Cannot steal token	Attack prevented (attacker cannot create an authorized referrer for its hardware request).	Attack prevented (attacker cannot steal a token, so he must get a new one from the device server, pretending to be from domain X; however, X already has an outstanding token, so device server rejects the new token request).
Can steal token	Attack prevented (attacker cannot create an authorized referrer for its hardware request).	Attack succeeds (attacker can steal token from X's page and put X in its referrer field, effectively masquerading as X).

(c) Gibraltar attack resilience (a legitimately authorized page from domain X is running).

Figure 5. Gibraltar security properties.

browser stole/fetched a token and acquired the data itself), neither Gibraltar nor HTML5 can prevent the browser from arbitrarily inspecting, modifying, or disseminating the data.

Suppose that, through clever engineering, the browser cannot be subverted by malicious web pages. Further suppose that the browser is trusted not to fake referrer fields, steal tokens from authorized domains, or otherwise subvert the Gibraltar access controls. Even in these situations, malicious web pages can still leak hardware data to remote servers. For example, suppose that the user has authorized domain `x.com` to access hardware, but not `y.com`. The same-origin policy ostensibly prevents JavaScript running on `http://x.com/index.html` from sending data to `y.com`'s domain. However, this security policy is easily circumvented in practice. For example, the JavaScript in `x.com`'s page can read the user's GPS data and create an iframe with a URL like `http://y.com/page.index?lat=LAT_DATA long=LON_DATA`. By loading the frame, the browser implicitly sends the GPS data to `y.com`'s web server.

If the browser is trusted, it can prevent such leakage by tracking the information flow between Gibraltar AJAX requests and externalized data objects like iframe URLs. This is similar to what TaintDroid [6] does, although TaintDroid tracks data flow through a Java VM instead of a browser.

If the browser is untrusted, we can place the taint tracking infrastructure outside of the browser, e.g., in the underlying operating system. However, regardless of where the taint tracker resides, it must allow the user to whitelist certain pairs of domains and hardware data. For example, suppose that the user has authorized only `x.com` to access the GPS unit. Whenever the data flow system detects that GPS data is about to hit the network, it must ensure that the endpoint resides in x.com's domain, e.g., by doing a reverse DNS lookup on the endpoint's IP address.

If the taint system performs that check, it can prevent data from directly leaking to unauthorized domains. However, a full security suite requires both a taint tracker *and* Gibraltar, since a taint tracker alone cannot prevent several damaging attacks. For example, if only a taint tracker is present, a misbehaving browser could send hardware data to an authorized domain even if the user is not currently viewing a page in that domain. Such persistent snooping is problematic because it lets the authorized domain build a huge database of contextual information about the user, even though the user only intended for that data to be collected when she was actually browsing a web page from that domain. As shown in Figure 5(b), Gibraltar detects this attack if the user has not opened a page for an authorized domain. This is because the browser cannot surreptitiously stream data without triggering a sensor widget. However, if the user does have an authorized page open, and is running a browser with weak memory isolation, Gibraltar cannot stop the stolen token attacks shown in the bottom-right corner of Figure 5(c). Note that HTML5 cannot stop any of these attacks, since it lacks sensor widgets or a method for assigning device ACLs to web pages.

Note that taint tracking and whitelists cannot prevent all kinds of information leakage. For example, a malicious browser can post sensitive data to a whitelisted site using a format that the site does not treat as sensitive. For example, user data could be encoded as a comment in a web forum. When combined with cross-site request forgery (CSRF), an attacker may be able to download the exfiltrated user data. Gibraltar is compatible with approaches for stopping CSRF attacks (e.g., [25, 33]).

6. Evaluation

In this section, we ask two fundamental questions about Gibraltar's performance. First, is an HTTP channel fast enough to support high frequency sensors and interactive applications? Second, is Gibraltar competitive with HTML5 in terms of performance?

As described in Section 3, we wrote device servers for two platforms. The first server runs on Android 2.2 phones, and we tested it on two handsets: a Nexus One with 512 MB of RAM and a 1 GHz Qualcomm Snapdragon processor, and a Droid X with 512 MB of RAM and a 1 GHz Texas Instruments OMAP processor. We also wrote a device server for Windows PCs. We tested that server on a Windows 7 machine with 4 GB of RAM and an Intel Core2 processor with two 2.66 GHz cores.

6.1 Access Latency

Multi-core machines: We define a device's *access latency* as the amount of time that a client perceives a synchronous device operation to take. Figure 6 shows access latencies for the hard disk on the dual-core desktop machine. Each bar represents the average of 250 trials, with each read or write involving 1 KB of data. HTML5 disk accesses were implemented using the DOM storage API [11], whereas Gibraltar disk accesses were handled by the device server and accessed a partitioned region of the local file system owned by the device server. All reads targeted prior write addresses, meaning that the reads should hit in the block cache inside the device server or the HTML5 browser.

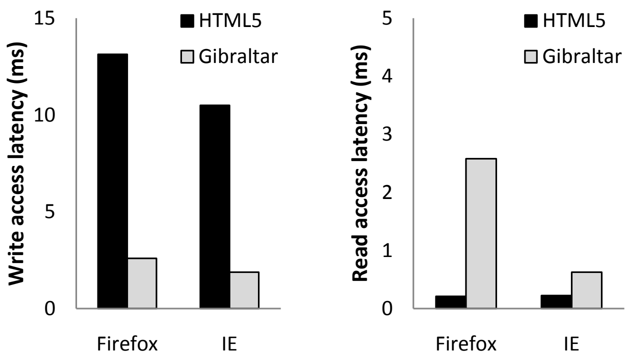


Figure 6. Read and write latencies to the hard disk on the dual-core desktop machine.

The absolute latencies for Gibraltar's disk accesses are small on both Firefox 3.6 and IE8. For example, a Gibraltar-enabled page on IE8 can read 1 KB of data with a latency of 0.62 ms; on Firefox, the page can perform a similar read with 2.58 ms of latency. While Gibraltar's read performance is worse than that of HTML5, it is more than sufficient to support common use cases for local storage, such as caching user data to avoid fetching it over a slow network.

For disk writes on both browsers, Gibraltar is more than five times faster than HTML5. This is because the Gibraltar device server asynchronously writes back data, whereas Firefox and IE have a write-through policy. Switching Gibraltar to a write-through policy would result in similar performance to HTML5, since the primary overhead would be mechanical disk latencies, not HTTP overhead.

Single-core machines: Our desktop machine had a dual-core processor, meaning that the device server and the web browser rarely had to contend for a core. In particular, once the device server had invoked a `send()` system call to transfer device data to the browser, the OS could usually swap the browser immediately onto one of the two cores. On a single core machine, the browser might have to wait for a non-trivial amount of time, since multiple processes besides the browser are competing for a single core.

Figure 7 depicts access latencies to the Null device on the Droid X phone (the Null device immediately returns an empty message). By using `setsocketopt()` to disable the TCP Nagle algorithm, we prod TCP into sending small packets immediately instead of trying to aggregate several

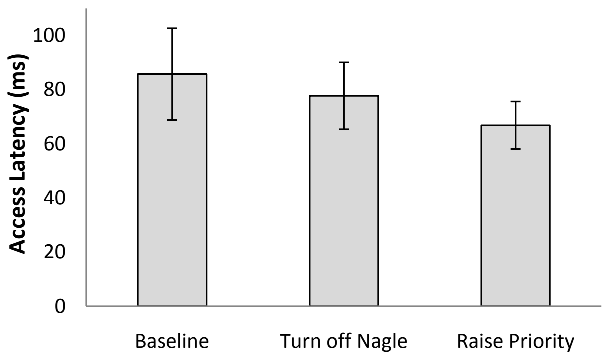


Figure 7. GibDroid Null-device access latencies (single-core phone). Error bars represent one standard deviation.

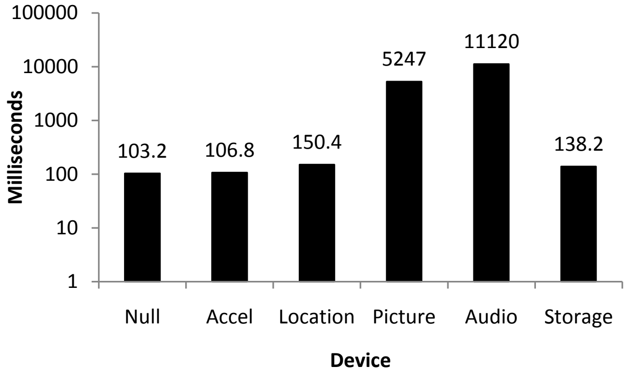


Figure 8. Nexus One access latencies (mobile browser accessing local hardware). Note that the y-axis is log-scale.

small packets into one large one. This decreases the average access latency from 87 ms to 78 ms; it also decreases the standard deviation from 34 ms to 25 ms. By raising the priority of the device server thread and the receiving browser thread, we can further decrease the latency to 67 ± 18 ms. However, the raw performance is still worse than in the dual-core case due to scheduling jitter. For example, looking at single-core results for individual trials, we saw access latencies as low as 29 ms, and as high as 144 ms.

Multi-core processors are already pervasive on desktop systems, and new mobile phones and tablets like the LG Optimus 2X have dual-core processors. Thus, we expect that scheduling jitter will soon become a non-issue for Gibraltar. In the rest of this section, we provide additional evaluation results using the single-core Nexus One phone. We show that even on a single-core machine, Gibraltar is fast enough to support interactive applications.

Accessing Sensors on the Nexus One: Figure 8 depicts the access latency for various devices on the Nexus One phone. The accelerometer and the GPS unit are the sensors that applications query at the fastest rate. Figure 8 shows that the accelerometer can be queried 9.4 times a second, and the

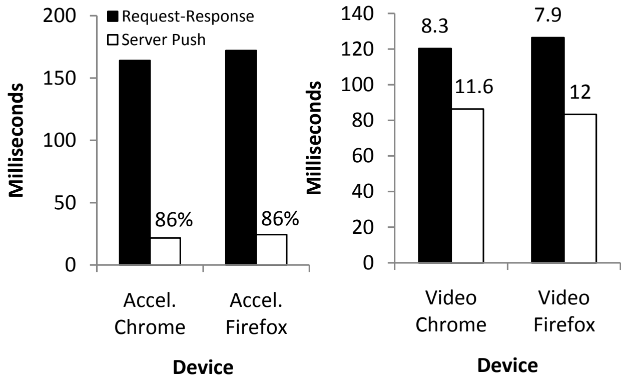


Figure 9. Nexus One access latencies (desktop browser accessing phone hardware). Top-bar numbers for accelerometer represent improvements in sample frequency; top-bar numbers for video represent frame rates.

GPS unit can be queried 6.6 times a second. As we discuss in Section 6.4, these sampling rates are sufficient to support games and interactive mapping applications.

Accessing the camera or the microphone through Gibraltar is much more expensive than accessing the accelerometer. However, most of the latency arises from the inherently expensive initialization costs for those devices. For example, GibDroid adds 160 ms to the inherent cost of sampling 10 seconds of audio data, and 560 ms to the inherent cost of taking a picture. In both cases, the bulk of Gibraltar's overhead came from the `Base64` encoding that the device server must perform before it can send binary data to the application.

The results in Figure 8 used the request-response version of the Gibraltar protocol. On browsers that support forever frames (§3), Gibraltar can use server-push techniques to decrease client-perceived access latencies to devices. Figure 9 quantifies this improvement for desktop browsers accessing phone hardware over a wireless connection. For example, for video on Firefox, frame access latencies decreased from 126 ms to 83 ms; this improved the streaming rate for live video from 8 frames per second to 12. For the accelerometer, access latencies decreased from 173 ms to 22 ms, allowing the client to fetch accelerometer readings at a rate of 45 Hz. This was close to the native hardware limit of 50 Hz. Note, however, that the performance gains in both cases arose not just from the server-push technique, but from the fact that the device server and the web browser ran on different machines (and thus different processors). This ameliorated some of the scheduling jitter that arises when the device server and the browser run on the same core.

6.2 Sampling Throughput

Low access latencies improve the freshness of the data that the client receives. However, the client may still be unable to receive data at the native sampling frequency. Thus, the device server continuously gathers information from high data

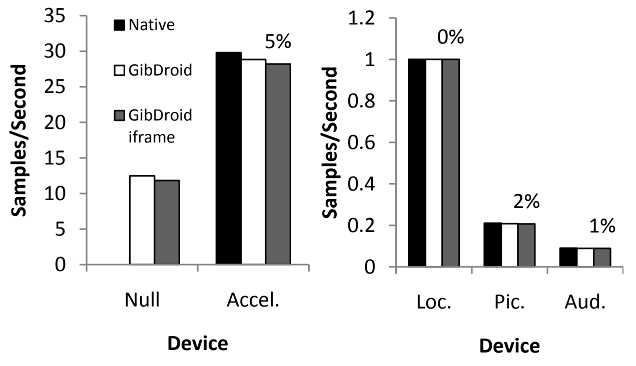


Figure 10. GibDroid sampling throughputs. Top-bar fractions show the percentage slowdown of Gibraltar with an iframe compared to native execution.

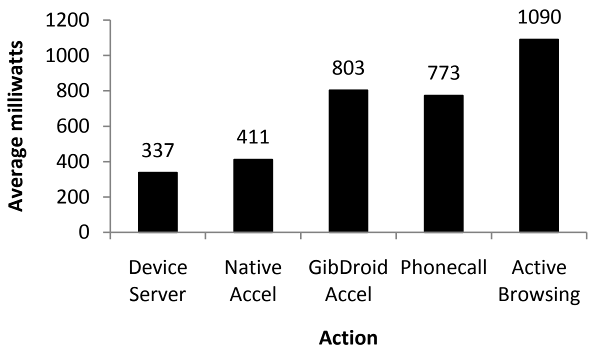


Figure 11. GibDroid power consumption.

rate devices like the accelerometer and the GPS unit. When the server gets a read request for such a device, it returns all of the data that has accumulated since the last query. Thus, an application can analyze the entire data stream even if it cannot access every sample at the native data rate.

Figure 10 depicts GibDroid's sampling throughput using the built-in Android browser to access phone hardware. Each bar represents the maximum number of data samples accessible per second to a native application, a Gibraltar page using an inner iframe (§2), and a Gibraltar page in which the outer iframe directly issues AJAX requests. Throughput degradation was less than 5% for all devices. Figure 10 also shows that cross-frame `postMessage()` overhead was minimal. Note that the accelerometer throughput was greater than the Null device throughput because GibDroid batched multiple accelerometer samples per HTTP response.

6.3 Power

On mobile devices, minimizing power consumption is extremely important. To measure Gibraltar's impact on battery life, we attached a Monsoon Power Monitor [20] to the Nexus One. The Monsoon acted as an energy source while simultaneously measuring how much power it transferred to the phone. We set the phone's screen brightness to the minimum setting and enabled the phone's "airplane mode" when running tests that did not involve radios.

Figure 11 shows power consumption in several different scenarios. The first bar depicts the power consumption for an idle device server. Running an idle server costs 337 mW, and this is essentially the base cost of having the phone turned on. Continuously querying the accelerometer in a native application requires 411 mW. In contrast, using GibDroid to continuously query the device costs 803 mW; however, this cost includes the power spent by the server *and* the browser. By comparison, making a phone call requires 773 mW, and actively browsing the Internet uses over 1W. Thus, we believe that Gibraltar's power usage is similar to that of other mobile applications.

6.4 Applications

For the final part of our evaluation, we examined the performance of the four Gibraltar-enabled applications that we described in Section 4. We evaluated all four applications on the GibDroid platform.

Our map application took an average of 64 ms to load a cached map tile, but 372 ms to fetch one from the Internet. This result is not surprising, since accessing local storage should be faster than pulling data across the wide area.

For our audio classification application, the key performance metric is how long the classification takes. For a 52 KB WAV file representing 10 seconds of data, feature extraction took approximately 6 seconds, and classification of the result took 1.5 seconds. These experiments used a JavaScript implementation of the classification algorithms. For larger audio files, the application could use Gibraltar's native computation kernels to boost performance.

We evaluated Paint and Pacman by running them on a Chrome desktop browser which communicated with GibDroid through a USB cable. Paint was able to sense 9.83 motions per second; this number is an application-level latency that includes the Gibraltar access latency and the overhead of updating the HTML `Canvas` object. Pacman had similar performance. In both cases, the phone was able to control the application with no user-perceived delay. We plan to run further tests over a wireless network which allows the phone to be untethered from the desktop.

7. Related Work

In Section 1, we described the disadvantages of using native code plugins like Flash to provide hardware access to web pages. We also described why HTML5 is a step in the right direction, but not a complete solution.

Like Gibraltar, Maverick [32] provides web pages with hardware access. Maverick lets web developers write USB drivers using JavaScript or NaCl. Maverick sandboxes each untrusted page and USB driver; the components exchange messages through the trusted Maverick kernel. Maverick differs from Gibraltar in three key ways. First, Maverick is limited to the USB interface, whereas Gibraltar's client-side JavaScript library can layer arbitrary hardware protocols atop HTTP. Second, unlike USB, HTTP provides straightforward support for off-platform devices. Third, Maverick does not have mechanisms like sensor widgets that detect misbehaving applications. Thus, Maverick cannot prevent buggy or malicious pages from using the driver infrastructure in ways that the user did not intend. Maverick does have better performance than the current implementation of Gibraltar since Maverick provides IPC via native code NaCl channels instead of via standard HTTP over TCP. However, with kernel support for fast-path localhost-to-localhost TCP connections, and/or NIC support for offloading TCP-related computations to hardware, we believe that Gibraltars performance can approach that of Maverick.

PhoneGap [26] is a framework for building cross-platform, device-aware mobile applications. A PhoneGap application consists of JavaScript, HTML, CSS, and a bundled chrome-less browser whose JavaScript runtime has been extended to export hardware interfaces. Like Gibraltar, PhoneGap allows developers to write device-aware applications using the traditional web stack. Compared to Gibraltar, PhoneGap has three limitations. First, PhoneGap's hardware interface is philosophically equivalent to the HTML5 interface, and thus has similar drawbacks with respect to interface and security. Second, a PhoneGap program is a native application and must be explicitly installed, unlike a Gibraltar web page. Third, PhoneGap applications run within the `file://` protocol, not the `http://` protocol. Thus, unlike Gibraltar web pages, PhoneGap programs are not restricted by the same domain policy. This allows a PhoneGap program to load multiple frames from multiple domains and manipulate their data in ways that would fail in the `http://` context and violate the security assumptions of the remote domains.

In Palm's webOS [1], applications are written in JavaScript, HTML, and CSS. However, these programs are not web applications in the standard sense—they rely on webOS' special runtime, and they will not execute inside actual web browsers. The webOS runtime is a customized version of the popular WebKit browser engine. It exposes HTML5-style device interfaces to applications, and thus suffers from the problems that we discussed in prior sections.

Microkernel browsers like OP [9] and Gazelle [36] restructure the browser into multiple untrusted modules that exchange messages through a small, trusted kernel. Gibraltar's device server is somewhat like a trusted microkernel which mediates hardware access. However, previous microkernel browsers do not change the hardware interface exposed to web pages, since these browsers use off-the-shelf JavaScript runtimes that export the HTML5 interface.

Several projects from the sensor network community expose hardware data using web protocols [5, 28, 38]. However, these systems do not address the security challenge of authenticating hardware requests that emanate from potentially untrustworthy browsers. Gibraltar also exports a richer interface for device querying and management.

8. Conclusions

Gibraltar's key insight is that web pages can access hardware devices by treating them like web servers. Gibraltar sandboxes the browser, shifts authority for device accesses to a small, native code device server, and forces the browser to access hardware via HTTP. Using this privilege separation and sensor widgets, Gibraltar provides better security than HTML5; the resulting API is also easier to program against. Experiments show that the HTTP device protocol is fast enough to support real, interactive applications that make frequent hardware accesses.

References

[1] M. Allen. *Palm webOS: The Insider's Guide to Developing Applications in JavaScript using the Palm Mojo Framework*. O'Reilly Media, Sebastopol, CA, 2009.

[2] J. Bartel and et. al. i-Jetty: Webserver for the Android mobile platform. `http://code.google.com/p/i-jetty`.

[3] D. Crane and P. McCarthy. *Comet and Reverse Ajax: The Next-Generation Ajax 2.0*. Apress, New York, NY, 2008.

[4] M. Daniel, J. Honoroff, and C. Miller. Engineering Heap Overflow Exploits with JavaScript. In *Proceedings of USENIX Workshop on Offensive Technologies*, 2008.

[5] W. Drytkiewicz, I. Radusch, S. Arbanowski, and R. Popescu-Zeletin. pREST: A REST-based Protocol for Pervasive Systems. In *Proceedings of IEEE International Conference on Mobile Ad-hoc and Sensor Systems*, October 2004.

[6] W. Enck, P. Gilbert, B. gon Chun, L. Cox, J. Jung, P. McDaniel, and A. N. Sheth. TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *Proceedings of OSDI*, 2010.

[7] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616 (Draft Standard), June 1999.

[8] D. Flanagan. *JavaScript: The Definitive Guide, Fifth Edition*. O'Reilly Media, Inc., Sebastopol, CA, 2006.

[9] C. Grier, S. Tang, and S. T. King. Secure web browsing with the op web browser. In *Proceedings of the 2008 IEEE Symposium on Security and Privacy*, pages 402–416, Washington, DC, USA, 2008. IEEE Computer Society.

[10] I. Hickson. HTML5: A vocabulary and associated APIs for HTML and XHTML. `http://dev.w3.org/html5/spec/`.

[11] I. Hickson. Web Storage: Editor's Draft, August 20, 2010. `http://dev.w3.org/html5/webstorage`.

[12] J. Howell and S. Schechter. What you see is what they get: Protecting users from unwanted use of microphones, cameras, and other sensors. In *Proceedings of W2SP*, 2010.

[13] Khronos Group. OpenCL. `http://www.khronos.org/opencl`.

[14] E. Koukoumidis, D. Lymberopoulos, J. Liu, and D. Burger. Improving Mobile Search User Experience with SONGO. Microsoft Research Tech Report MSR-TR-2010-15, February 18, 2010.

[15] N. Landsteiner. How to Write a Pacman Game in JavaScript. `http://www.masswerk.at/JavaPac/`

pacman-howto.html.

[16] H. Lu, W. Pan, N. D. Lane, T. Choudhury, and A. T. Campbell. Soundsense: scalable sound sensing for people-centric applications on mobile phones. In *MobiSys '09*, pages 165–178, New York, NY, USA, 2009. ACM.

[17] MapQuest. Mapquest maps. http://www.mapquest.com/.

[18] L. Masinter. The "data" URL Scheme. RFC 2397 (Draft Standard), August 1998.

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

[20] Monsoon Soluitions Inc. Monsoon power monitor. https://www.msoon.com/LabEquipment/PowerMonitor/.

[21] R. Naraine. Pwn2Own 2010: iPhone hacked, SMS database hijacked. ZDNet. http://www.zdnet.com/blog/security/pwn2own-2010-iphone-hacked-sms-database-hijacked/5836, March 2010.

[22] Nike. Nike+. http://nikerunning.nike.com/nikeos/p/nikeplus/en_US/.

[23] K. Noyes. Android Browser Flaw Exposes User Data. PC-World. https://www.pcworld.com/businesscenter/article/211623/android_browser_flaw_exposes_user_data.html, November 24 2010.

[24] I. Oksanen and D. Hazael-Massieux. HTML Media Capture. http://www.w3.org/TR/html-media-capture/.

[25] OWASP (The Open Web Application Security Project. CSRF-Guard. https://www.owasp.org/index.php/Category:OWASP_CSRFGuard_Project.

[26] Phonegap. PhoneGap. http://www.phonegap.com/.

[27] A. Popescu. Geolocation API specification. http://dev.w3.org/geo/api/spec-source.html.

[28] B. Priyantha, A. Kansal, M. Goraczko, and F. Zhao. Tiny web services for sensor device interoperability. In *Proceedings of IPSN*, pages 567–568, Washington, DC, USA, 2008. IEEE Computer Society.

[29] N. Provos, M. Friedl, and P. Honeyman. Preventing Privilege Escalation. In *Proceedings of USENIX Security*, 2003.

[30] P. Ratanaworabhan, B. Livshits, and B. Zorn. NOZZLE: A Defense Against Heap-spraying Code Injection Attacks. In *Proceedings of USENIX Security*, 2009.

[31] C. Reis and S. Gribble. Isolating Web Programs in Modern Browser Architectures. In *Proceedings of EuroSys*, Nuremberg, Germany, April 2009.

[32] D. Richardson and S. Gribble. Maverick: Providing Web Applications with Safe and Flexible Access to Local Devices. In *Proceedings of USENIX WebApps*, Boston, MA, June 2011.

[33] P. D. Ryck, L. Desmet, T. Heyman, F. Piessens, and W. Joosen. CsFire: Transparent Client-side Mitigation of Malicious Cross-domain Requests. In *International Symposium on Engineering Secure Software and Systems*, February 2010.

[34] Secunia. Secunia Advisory SA29787: Mozilla Firefox Javascript Garbage Collector Vulnerability. http://secunia.com/advisories/29787, April 21 2008.

[35] P. Vixie. Dynamic Updates in the Domain Name System (DNS Update). RFC 2136 (Draft Standard), April 1997.

[36] H. J. Wang, C. Grier, A. Moshchuk, S. T. King, P. Choudhury, and H. Venter. The Multi-principal OS Construction of the Gazelle Web Browser. In *Proceedings of USENIX Security*, pages 417–432, 2009.

[37] World Wide Web Consortium (W3C). Access Control for Cross-Site Requests. W3C Working Draft, September 12 2008.

[38] D. Yazar and A. Dunkels. Efficient Application Integration in IP-Based Sensor Networks. In *Proceedings of BuildSys*, November 2009.

[39] A. R. Yumerefendi, B. Mickle, and O. P. Cox. Tightlip: Keeping applications from spilling the beans. In *Proceedings of NSDI*, 2007.