Maverick: Providing Web Applications with Safe and Flexible Access to Local Devices

David W. Richardson and Steven D. Gribble
Department of Computer Science & Engineering
University of Washington

Abstract

Web browsers do not yet provide Web programs with the same safe, convenient access to local devices that operating systems provide to native programs. As a result, Web programmers must either wait for the slowly evolving HTML standard to add support for the device classes they want to use, or they must use difficult to deploy browser plug-ins to add the access they need.

This paper describes Maverick, a browser that provides Web applications with safe and flexible access to local devices. Maverick lets Web programmers implement USB device drivers and frameworks, like file systems or streaming video layers, using standard Web programming technologies such as HTML, JavaScript, or even code executed in a native client sandbox. These Web drivers and Web frameworks are downloaded dynamically from Web servers and executed by browsers alongside Web applications. Maverick provides Web drivers with protected access to the USB bus, and it provides Web drivers and frameworks with event-driven IPC channels to communicate with each other and with Web applications.

We prototyped Maverick by modifying the Chrome Web browser and the Linux kernel. Using Maverick, we have implemented: several Web drivers, including a USB mass storage driver and a Webcam driver; several Web frameworks, including a FAT16 filesystem and a streaming video framework; and, several Web applications that exercise them. Our experiments show that Web drivers, frameworks, and applications are practical, easy to author, and have sufficient performance, even when implemented in JavaScript.

1 Introduction

Web browsers do not yet provide Web programs with the same safe, convenient access to local devices that OSs provide to native programs. Digital devices like cameras, printers, scanners, smartphones, and GPS trackers are increasingly pervasive, yet browsers currently provide little support to Web applications for accessing them. The support that does exist is limited to a handful of HTML tags for accessing a small number of common device classes, such as Webcams or microphones.

Today, Web programmers that want to use unsupported or exotic local devices must either wait for HTML standards to evolve to include them, or they must implement, deploy, and support browser plug-ins that users may be reluctant to install. Such poor choices limit the functionality of Web applications and discourage the development and adoption of new, interesting local devices.

In this paper, we describe Maverick, an experimental browser that gives Web applications safe and flexible access to local USB devices. Maverick takes the aggressive approach of removing the responsibility for managing devices and device frameworks from the host operating system and empowering the browser to execute device drivers and frameworks alongside Web applications. Specifically, instead of requiring users to install USB device drivers into their host OS, Maverick dynamically finds, downloads, and executes *Web drivers* that are written with standard Web programming technologies like HTML, JavaScript, or Native Client (NaCl) [24], and that directly communicate with the USB devices they drive.

Similarly, instead of relying on device frameworks within the host OS, such as file systems or video frameworks, Maverick finds, downloads, and executes *Web frameworks* to provide Web applications with convenient high-level abstractions. Maverick permits Web applications to communicate directly with Web frameworks, which in turn communicate with Web drivers.

Maverick's approach has several advantages. Web developers can add support for new USB devices and make them immediately accessible to any Maverick user and Web application without waiting for updates to slowly moving standards bodies, browser vendors, or operating systems. Maverick also inherits many of the safety and security benefits of running drivers in user-level, such as insulating the OS from driver bugs.

Maverick addresses three main challenges:

1. **Security.** Maverick must isolate Web drivers and Web frameworks to prevent access to unauthorized devices or interference with unrelated Web applications and native software. Maverick uses existing JavaScript and NaCl sandboxes to contain Web drivers and frameworks. In addition, Maverick exposes a virtualized USB bus to Web drivers, granting each driver the ability to send and receive USB messages only to the devices for which it is authorized. So that Web drivers, frameworks, and applications can interact flexibly and efficiently, Maverick provides them with protected event-driven IPC channels.

2. **Performance.** Web drivers and frameworks must be efficient. We prototyped Maverick by modifying the Chrome browser and the Linux kernel, implemented several drivers and frameworks in both JavaScript and NaCl, and compared them to native Linux equivalents. Not surprisingly, user-level drivers in general, and JavaScript drivers and frameworks in particular, are significantly slower than their Linux counterparts. Nonetheless, our experiments demonstrate they are fast enough to support many interesting USB devices and applications. As well, we show that NaCl can achieve performance closer to that of in-kernel drivers and frameworks.

3. **Usability**. Maverick must avoid burdening users with making confusing and error-prone decisions on how to select trustworthy and compatible drivers and frameworks for Web applications. To do this, Maverick allows users to configure their browsers to trust one or more Maverick *domain providers.* A domain provider is a trusted third-party like Google or Microsoft that is responsible for selecting and bundling together a set of interoperable Web frameworks and drivers.

We have prototyped several Web drivers and frameworks and applications that exercise them, including: a USB mass storage driver, a Webcam driver, a FAT filesystem framework, and a streaming video framework. Overall, our experience with Maverick suggests that Web drivers and frameworks are straightforward to implement, and are safe and practical from a performance, security, and usability standpoint.

The rest of this paper is structured as follows. In Section 2, we present a brief overview of USB. Section 3 describes the architecture of Maverick, and Section 4 presents our prototype implementation. In Section 5, we evaluate the performance and security of Maverick and we showcase several applications. After discussing related work in Section 6, we conclude.

2 A Brief Overview of USB

Maverick exposes USB devices to Web drivers and applications. We chose to focus on USB for two reasons; first, it has become the predominant interconnect for most consumer devices, making it an attractive target. Second, since USB is message-oriented, it was relatively straightforward to expose USB message transmission and reception to JavaScript and NaCl. In contrast, we believe it would be much less natural to expose complex device interconnects, such as PCI, that use more architecturally-dependent features like DMA and memory-mapped I/O.

In this section, we provide a brief overview of USB device abstractions and protocols. Readers familiar with USB may choose to skip to Section 3.

2.1 USB Devices and Communication Channels

A USB bus connects a host, such as a laptop or desktop, to multiple peripheral devices over a star topology. Some USB devices consist of more than one *logical device*; for example, a Web camera might consist of a video camera and a microphone packaged together into a single physical box. Each of these logical devices would appear as a separate addressable entity on the USB bus.

A logical device consists of one or more communication *endpoints* associated with some specific function of the logical device. A host establishes a *pipe* to an endpoint to communicate with it. There is a one-to-one mapping between pipes and endpoint. Each endpoint and its corresponding pipe are typed. USB supports four kinds of pipes:

- *Control*. A control pipe facilitates the bidirectional exchange of small control messages used to query or control a device. The USB specification mandates that hosts must reserve 10% of the USB bus bandwidth for control traffic. All devices have at least one control endpoint.

- *Interrupt*. An interrupt pipe is a unidirectional channel used to convey messages from a device to a host. For example, USB keyboards generate interrupt messages when key press events occur. Interrupt messages are latency sensitive; hosts must poll interrupt endpoints sufficiently frequently to ensure responsiveness.

- *Isochronous*. An isochronous pipe is a unidirectional channel used to transfer a continuous stream of data, such as video frames or audio packets. Hosts must schedule USB messages to provide a guaranteed amount of bandwidth to an established isochronous pipe. Isochronous pipes may experience occasional data loss.

- *Bulk.* Bulk pipes are unidirectional channels that provide reliable data transfer but no bandwidth guarantees. USB bulk storage devices and printers often use bulk pipes.

2.2 Host OS Abstractions and Duties

USB bus bandwidth is allocated and scheduled into time slices called *frames*. For high speed USB 2.0 devices, frames have a 125 micro-second interval, permitting up to 8,000 frames per second. The host operating system is responsible for scheduling USB packets into frames. Because USB mandates reserved bandwidth for isochronous pipes and responsiveness for interrupt pipes, the OS must queue USB packets, potentially delaying some to meet the scheduling demands of isochronous and interrupt traffic. Packets associated with bulk and control transfers are scheduled whenever a frame has available bandwidth not already consumed by interrupt or isochronous transfers.

Operating systems typically abstract USB transactions into a data structure called a USB request block (URB). A URB encapsulates a single, asynchronous interaction between the host and a device. The URB structure accommodates a request, data to be transferred, and a completion status message. Device drivers allocate and submit URBs to low-level, device-independent USB processing code within an OS. The OS schedules the transmission of messages and data indicated by the URB, and upon completion, notifies the driver.

While programmers can manually construct URBs for any type of data transfer, operating systems typically supply USB libraries with higher-level abstractions for constructing, sending, and receiving URBs based on the specified type of data transfer.

3 Architecture

Maverick splits a computer into two worlds (Figure 1): a legacy *desktop world* that contains the underlying host operating system, its drivers and frameworks, and applications, and the *Maverick world* that executes on top of a browser. Each world is isolated from the other with respect to the USB devices they can access: a given device is assigned to one of the two worlds, and the other cannot observe or influence it.

The assignment of USB devices to worlds is managed by the *USB world splitter* in the host OS. When a new USB device is detected, the splitter prompts the user

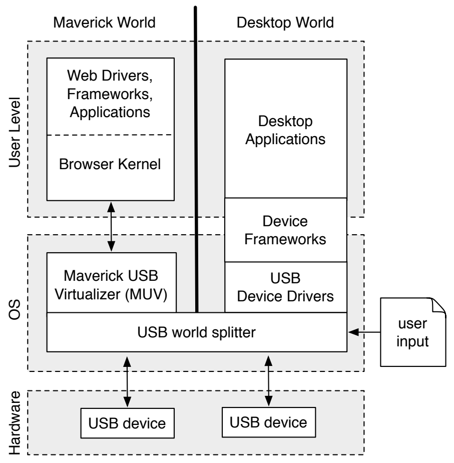

Figure 1: **Maverick "World Splitting."** Maverick splits computers into the legacy desktop world and the Maverick web world. USB devices are partitioned between the two worlds; each world runs its own drivers, frameworks, and applications.

to assign the device to either the desktop or Maverick world. The splitter then enforces this assignment when routing messages between devices and the two worlds.

3.1 The Desktop World

The desktop world exists to facilitate incremental deployment and backwards compatibility. Users can run unmodified legacy device drivers, frameworks, and applications in it: besides the presence of the USB world splitter, the desktop world is unaware of the existence of the Maverick world. Thus, in the desktop world, USB device drivers are installed into the host OS and communicate with devices using the USB core libraries provided by the OS. Drivers abstract away the details of specific devices and interfaces with frameworks, such as file systems, network stacks, and video frameworks. A framework, which is typically implemented partially in the host OS kernel and partially as sets of user-mode libraries, provides high-level, device-independent abstractions to applications.

3.2 The Maverick World

The Maverick world has some similarity to the legacy desktop world, in that the structural relationship between drivers, frameworks, and applications is the same in both cases. However, unlike the legacy world, in Maverick these components are (1) dynamically downloaded rather than installed, (2) implemented using Web programming technologies such as JavaScript and NaCl, and (3) executed by a browser kernel entirely at the user-level.

Figure 2 shows the architecture of the Maverick world.

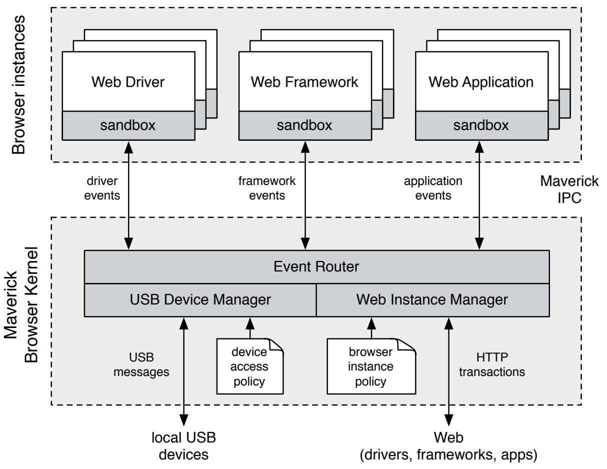

Figure 2: **The "Maverick World."** The Maverick world consists of a trusted browser kernel and untrusted browser instances. The kernel manages instances, provides IPC channels, and relays USB messages between authorized drivers and local USB devices.

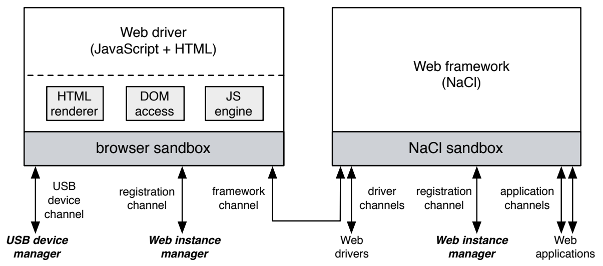

Figure 3: **Maverick Web Browser Instances.** Web drivers, frameworks, and applications execute inside browser instances. Instances contain JavaScript+HTML or NaCl code, and they interact with each other and the browser kernel via IPC channels.

At a high-level, the world is deconstructed into two components: untrusted *browser instances* and the trusted *browser kernel*. We describe each component below.

3.2.1 Maverick Browser Instances

Maverick browser instances contain untrusted code provided by a remote Web service. Browser instances are sandboxed from each other, the browser kernel, and the legacy desktop world. Instances can implement device driver functionality, framework functionality, or application-level functionality using standard Web programming technologies. We focus on two variants: JavaScript and NaCl instances (Figure 3). Similar to browsers like Chrome [20], a JavaScript instance contains DOM bindings, a JavaScript interpreter, and an HTML renderer, in addition to the browser instance's code itself. A NaCl instance contains x86 code that is verified and contained by the NaCl sandbox.

When an instance is instantiated, it has access to a registration IPC channel provided to it by the browser kernel. When the instance has initialized itself, it uses the registration channel to alert the browser kernel, which then establishes point-to-point IPC channels between the instance and other instances with which it must communicate. We discuss the policy by which the browser kernel interconnects devices, frameworks, and applications in Section 3.3.

3.2.2 The Maverick Browser Kernel

The Maverick browser kernel serves two main roles. First, it provides the standard functions of a "typical" browser kernel, including: managing standard Web storage like cookies, cache, and history; providing network access to browser instances; enforcing the same-origin policy; and, implementing the browser's user interface [20]. We have not modified these functions, and will not describe them further in this paper.

Second, the browser kernel provides Web drivers with safe access to local USB devices and facilitates secure communication between drivers, frameworks, and applications. The Maverick browser kernel consists of three modules: (1) the USB device manager, (2) the Web instance manager, and (3) the event router. The USB device manager stores device information such as vendor and device IDs for devices made available to Maverick via the world splitter. As well, the device manager establishes the channel between a Web driver and its local USB device: messages transmitted by a driver are routed to device manager, which verifies that they are properly formatted and addressed to the driver's authorized device before relaying them down to the host OS.

The Web instance manager downloads and instantiates Web driver and framework instances. The Web instance manager is also responsible for establishing the point-to-point IPC channels between drivers, frameworks, and applications. The IPC channels are managed by the event router: each channel implements a reliable, point-to-point FIFO queue. Browser instances communicate over these channels using events that contain a name, untyped variable-length payload, and destination. Within these constraints, browser instances are free to define and use any communication protocol.

3.2.3 The Maverick USB Virtualizer

The Maverick USB Virtualizer (MUV) is a device-agnostic USB driver that lives inside the host OS. It virtualizes the USB API provided by the core USB libraries in the kernel, translating and packaging up USB messages from the kernel's USB core into events that delivered to the browser kernel, and from there, routed to the appropriate Web driver. USB events sent from Web drivers are routed from the browser kernel to the MUV and translated into appropriate calls into the host OS's USB core.

3.3 Naming and Binding of Web Drivers and Frameworks

As previously mentioned, Maverick must decide how to select, download, and interconnect Web driver and framework instances to applications. The policies it uses to do this are critical, as they impact safety, reliability, and compatibility: the policy must prevent users from being exposed to malicious drivers and frameworks, ensure that the set of drivers and frameworks that are instantiated are compatible with each other and with the applications, and that the applications can depend on a stable, coherent set of framework abstractions and interfaces.

We have experimented with several policies, and we describe two below. However, we feel that we are still just beginning to explore this topic: there are still many difficult and interesting research challenges to solve.

Application-driven. Under this policy, each Web application declares to Maverick the URLs of the frameworks that it requires. Similarly, each instantiated framework declares to Maverick the URLs of the drivers that it trusts and is compatible with. When a user navigates to an application, Maverick instantiates its declared frameworks, observes which drivers the framework is willing to have loaded, and identifies the subset of drivers that match available, underlying USB devices. Before the drivers are loaded and bound to USB devices, Maverick prompts the user for authorization: the prompt informs the user of the URLs of the application, frameworks, drivers, and devices that are involved.

This policy is simple, but has many disadvantages. First, since different applications may select different frameworks, there is no opportunity for frameworks (and their drivers) to form a coherent, interoperable set of abstractions. Two different applications may cause vastly different frameworks to load, and those frameworks will have no basis for interoperating or sharing the underlying devices between applications. Second, users likely have no basis to make reasonable authorization decisions: users know they want to use an application, but they cannot know whether the frameworks and drivers selected are trustworthy, especially since URLs in and of themselves are not particularly informative.

Domain-driven. Under this policy, users can configure their browsers to trust one or more Maverick *domain providers.* A domain provider is a trusted third-party that is responsible for selecting and bundling together a set of interoperable Web frameworks and drivers. Users name domain providers by URL; the document behind this URL contains the list of authorized framework and driver URLs. We anticipate that organizations like Google, Microsoft, or the FSF could be domain providers.

When loaded, an application declares to Maverick the domain provider it wants to use, and the frameworks within that domain that it requires. If the user has authorized the specified domain provider, Maverick will demand load the required frameworks. Once loaded, the framework declares the Web drivers that it requires; Maverick verifies that they are on the domain's authorized list, and loads them if so. Frameworks are responsible for prompting users before granting a Web application access to particular device or device class.

A given framework within a domain is loaded only once. Multiple applications that use the same domain are bound to that single instance, permitting the frameworks to facilitate sharing across those applications, as appropriate. Of course, this implies that the frameworks must provide adequate protection as well!

The benefit of the domain-driven approach is that the user makes a single higher-level trust decision and delegates to the trusted domain provider the responsibility for selecting appropriate, safe, and interoperable frameworks and drivers. As well, the domain provider can engineer its frameworks to be interoperable with each other, providing much of the same kind of API coherence, resource sharing, and protection that today's operating systems provide to applications. A disadvantage of this approach is that it may cause the user to place too much trust in a small number of domain providers, and it could lead to significant fragmentation of applications across domain providers.

4 Implementation

We implemented a prototype of Maverick by modifying the Chrome Web browser and the Linux kernel. Since Maverick uses Chrome, we inherit its process model: each browser instance (driver, framework, or application) executes in its own, separate Linux process. As well, Chrome's browser kernel executes as its own separate, trusted process. Rather than attempting to integrate our Maverick browser kernel components into Chrome's browser kernel, we bundled them into their own trusted process.

Using our prototype, we implemented several Web drivers, frameworks, and applications, both in JavaScript and in NaCl. In this section, we describe our aspects of the implementation, focusing on non-obvious issues.

4.1 Event Framework

Maverick browser instances communicate with each other and with the browser kernel using an asynchronous event-driven model, facilitating a natural integration with today's event-driven Web programming languages and browser abstractions. Our events are untyped, meaning that Web drivers, frameworks, and applications can exchange arbitrary data with each other, giving them the flexibility to define and implement their own high-level interfaces and protocols.

Maverick events have three fields: the name of the event, unstructured data payload, and a routing target address. The sender provides each of these fields, specifying a routing target of muv, $driverID$, $frameworkID$, or $applicationID$. This target address tells Maverick to route the event to either the MUV (via the USB device manager), or to a Web driver, framework, or application with the provided ID. The Maverick event router maintains routing tables to determine whether the sender has a valid communication channel to the target Maverick instance with the specified ID. If so, Maverick routes the event to the target's event queue.

4.1.1 Events in NaCl Instances

A programmer might choose to implement a Web driver, framework, or application using NaCl. To integrate our event framework into NaCl, we had to solve three problems. First, we needed to create IPC channels between NaCl instances and our Maverick browser kernel process; we implemented UNIX domain sockets as our channel transport. Second, we needed to be able to marshal Maverick events over that transport; we created a RPC layer using Google's protobuf library to do this [9].

Third, we needed to exchange events with the Web instance code. To do this, we extended the NaCl system call interface to permit sandboxed code to send events over the IPC channel, and to synchronously receive the next event from the channel. We also provide instance programmers with an (untrusted by Maverick) support library that spawns a NaCl thread to loop, issuing blocking receives on the IPC channel and dispatching events to a programmer-supplied callback function. This library provides programmers with convenient `createEvent()` and `postEvent()` functions, as well as functions for registering event handlers.

4.1.2 Events in JavaScript Instances

JavaScript programmers that want to use Maverick can include a convenience JavaScript library that we supply. This library provides an API that is syntactically similar to the API provided by our NaCl support library. This simplifies porting a NaCl Web driver to JavaScript, and vice-versa. Specifically, our library implements the following functionality:

1. To implement the IPC channel to the browser kernel, the library includes a hidden NaCl component that implements the IPC mechanisms we described above. We could have instead modified Chrome's JavaScript interpreter to expose this IPC channel, eliminating the need for NaCl support in the browser, but for the sake of simplicity we chose to leverage our existing NaCl code.
2. To deliver events to the instance programmer's JavaScript, we take advantage of the NPAPI interface provided by NaCl to invoke a callback handler on an object exposed to the instance through the DOM. The library creates a separate DOM object for each channel made available to the Web instance by the browser kernel.
3. The JavaScript library provides the programmer with convenience routines for base64 encoding and decoding binary data within event payloads, as well as protobuf support for exchanging structured messages with other browser instances.

4.2 The Maverick Browser Kernel

As previously mentioned, we implemented the trusted Maverick browser kernel to run as a separate process, independent of the Chrome browser kernel. At its heart, the event router component within the Maverick browser kernel is a threaded RPC server that establishes IPCs to browser instances and the MUV, and processes and routes events between them. We implemented the kernel in C++; for each browser instance, we allocate an event queue and spawn dedicated send and receive threads.

The Web instance manager component defines and processes browser instance registration events on behalf of the browser kernel. When a browser instance begins executing, it is expected to send one of these events over its registration channel. Similarly, the USB device manager component defines and processes USB message events, relaying them between authorized Web drivers and the MUV. Web drivers can create control or bulk URBs, start or terminate isochronous streams, and receive isochronous stream data (see Section 2). We have not yet implemented support for USB interrupt messages, as our experimental drivers have not required them, but doing so would be simple.

4.3 The Maverick USB Virtualizer and World Splitter

The Maverick world splitter is implemented as a dynamically loaded Linux USB device driver. When a USB device is attached to the host, the world splitter prompts the user to decide whether to associate the device with Maverick or not. For devices associated by the user with Maverick, the Linux USB subsystem relays the device's hardware IDs to the browser kernel which stores these tags in a registered device ID table that can later be used to bind compatible Web drivers to registered devices. The world splitter then routes all received USB callbacks up to the Maverick USB virtualizer. Devices not associated with Maverick by the user are bound to native Linux device drivers.

For every attached USB device, the MUV spawns a kernel thread devoted to handling that device. This thread maintains an RPC connection to the Maverick browser kernel, which it initially uses to register the device's vendor and product IDs. USB events are shuttled between the browser kernel and the MUV over these RPC connections; we ported a subset of protobuffer support into the Linux kernel to marshal events over these connections. The MUV dispatches events to the Linux core USB library.

To optimize the transmission of isochronous data from the Linux kernel to a Web driver, the MUV shortcuts the process of sending an isochronous URB to the device. When an isochronous URB completes, the MUV packages up the URB into a *completed_urb* event and sends it to the Web driver via the browser kernel. The MUV then immediately submits the next isochronous URB on behalf of the Web driver. As we show in Section 5, this optimization helps to improve bandwidth for streaming devices by eliminating some of the additional USB latency added by Maverick.

4.4 Example Drivers, Frameworks, and Applications

We have implemented several experimental Web drivers and frameworks. We built two versions of each, one in JavaScript and one in C++ using NaCl. Note that due to JavaScript's lack of raw data support, raw data in JavaScript drivers and frameworks are formatted as base64-encoded strings:

- **USB mass storage driver.** The USB mass storage specification consists of a "bulk-only" transport protocol used to initialize a device, exchange data, and handle error conditions. The SCSI command protocol is layered on top of this; SCSI commands are embedded inside bulk-only protocol messages. Our drivers implement enough of the SCSI protocol to initialize attached drives, probe for capacity, and read and write blocks. We have primarily experimented with SanDisk USB flash drives.
- **Logitech C200 Webcam driver.** Our Webcam drivers interact with the Logitech camera using Logitech's proprietary protocol; we ported a subset of this protocol using a Linux driver as reference. The drivers extract 320x240 resolution video frames at a rate of 30 frames per second over an isochronous stream, and post events containing raw frame data to an attached video framework.
- **FAT16 file system framework.** We implemented FAT16 compatible filesystem frameworks that expose file create, read, write, and delete operations to attached Web applications. The frameworks are also able to format and partition flash drives.
- **Video stream rendering framework.** Our video frameworks first convert raw frame data into a JPEG image; this only requires reformatting the frame data by adding an appropriate JPEG header. Next, the frameworks encode the images into base64 data URI strings [12], and post events containing them to attached Web applications. Web applications can then blit JPEGs to the screen by updating an HTML image tag's source attribute with the received data URI string.

To test these drivers and frameworks, we wrote a video streaming Web application called PhotoBooth. PhotoBooth allows users to view live video from an attached Webcam, capture the Webcam's live video stream and save it to a flash drive, or read and display a previously captured video stream from a flash drive.

5 Evaluation

In this section, we examine three aspects of our Maverick prototype: (1) its performance, (2) its security implications, and (3) its suitability for building device-enabled Web applications. We have not yet optimized our prototype implementation, so performance numbers should be considered an upper bound for a Maverick system.

5.1 Performance

We evaluate the performance of our prototype in two parts. First, to understand the overhead of moving device drivers out of the kernel and into the Web browser, we provide microbenchmarks that compare the performance of Web drivers to their native Linux drivers. Then, we evaluate the end-to-end performance of Maverick Web applications. Our measurements were gathered on an 8-core, 2GHz Intel Xeon machine with 6GB of RAM, running our modified Linux kernel.

5.1.1 Microbenchmarks

Our first set of experiments quantify the latency, throughput, and CPU utilization of Maverick Web drivers. To do this, we constructed a series of microbenchmarks that exercise both the JavaScript and NaCl versions of our

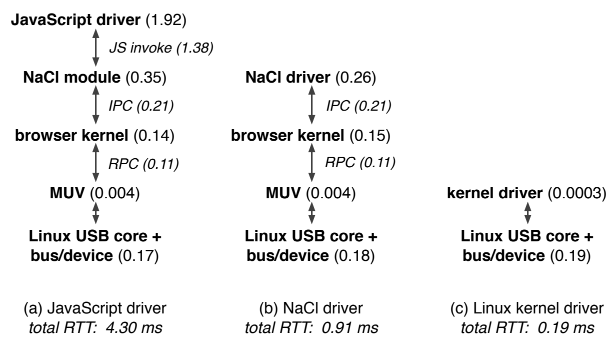

Figure 4: **Driver Latency.** The event flow and latencies of transacting a USB bulk URB with a USB flash drive, using (a) a JavaScript driver, (b) a NaCl driver, and (c) a Linux kernel driver. Numbers in parenthesis indicate time spent in that component or channel, in milliseconds.

storage and Webcam drivers. For a fair comparison, we also back-ported our NaCl drivers to run as native Linux drivers.

To measure the latency of performing USB operations from Web drivers, we instrumented our system to measure the time spent in various Maverick components when sending a single bulk message URB to a USB flash drive and receiving the response. Figure 4 depicts the involved components, the time spent in them, and the total round-trip time between the driver and device. Results are reported for each of the JavaScript, native client, and kernel drivers, averaged over 600 trials.

Not surprisingly, the JavaScript driver has the highest total latency. Its dominant factors are the latency of dispatching an event from the NaCl glue to JavaScript and the JavaScript driver execution itself. The NaCl glue overhead could be eliminated by modifying Chrome's JavaScript interpreter to deliver events rather than using our indirect NaCl route. The NaCl driver avoids these sources of overhead, but is roughly four times slower than the kernel driver due to the marshaling and transport of events from the Linux kernel to the browser.

To measure Web driver throughput and CPU utilization, we tested whether our Logitech C200 drivers were efficient enough to keep up with the isochronous URB transaction rate of 250 URB/s corresponding to a frame rate of 30 frame/s. For this experiment, we isolated driver performance by modifying our drivers to simply receive and drop URBs without further processing. Table 1 shows the URB rate sustained by each of our drivers, as well as the total CPU utilization (out of 8 cores). Only the JavaScript driver is unable to sustain the full streaming rate, as it saturates a single core, resulting in a loss of roughly one frame per second. However, future improvements to JavaScript execution engines should easily nudge this driver above the required 250 URB/s.

	sustained URB rate	CPU utilization
JavaScript Web driver	240 URB/s	13.4%
NaCl Web driver	250 URB/s	2.7%
Linux kernel driver	250 URB/s	0.25%

Table 1: **USB Webcam Driver Throughput.** This table shows the URB transaction rate that each driver could sustain, as well as the CPU utilization of the machine. The camera emits isochronous messages at a rate of 250 URBs per second.

5.1.2 End-to-End Performance

We now evaluate the end-to-end performance of Web applications in Maverick. Our application benchmarks exercise two drivers (USB storage and Webcam) and two frameworks (FAT16 and video). As before, we will compare using JavaScript, NaCl, and native kernel versions of the drivers.

To measure end-to-end latency, we instrumented the system to measure time spent in each major Maverick component when sending a storage operation from PhotoBooth and receiving a response. These components consist of the application, framework, driver, and the rest of Maverick. Time spent in Maverick includes all IPC and RPC channels, the browser kernel, the Linux kernel, and the USB bus and device itself. All reported measurements represent the average of 600 trials.

Table 2 shows our results. Not surprisingly, the JavaScript stack has the highest latency. Similar to our microbenchmark results, this overhead is mostly due to the 10 expensive IPC crossings between native client and JavaScript that are required because our prototype does not implement event delivery directly in Chrome. The JavaScript driver is also slow at processing byte-level data. The native client stack avoids most of these IPC overheads, but is still slower than the native desktop stack because of the cost of routing events from the kernel into user-space drivers and frameworks.

To measure Maverick's end-to-end throughput and CPU utilization, we benchmarked file create+write and file read operations on 20KB-sized files in PhotoBooth with the FAT16 framework and USB flash driver. As in earlier experiments, we compare three cases: the PhotoBooth application driving a JavaScript framework and a JavaScript driver, PhotoBooth driving a NaCl framework and a NaCl driver, and a native C application that issues similar file workload as PhotoBooth driving an in-Linux-kernel combined framework and driver. Our benchmarks leveraged Maverick's ability to run drivers and frameworks in parallel in separate Chrome processes and pipelines file system operations through them.

Table 3 shows our results. For all three versions, the file create+write benchmark achieved lower throughput than file reads, since file creation and file append both re-

PhotoBooth latency	time spent in component				end-to-end latency
	application	framework	driver	Maverick	
JavaScript driver/framework	0.8ms	1.7ms	3.3ms	8.7ms	14.5ms
NaCl driver/framework	0.8ms	0.35ms	0.44ms	3.1ms	4.7ms
native "app/FW" and Linux kernel driver	0.05 ms ("app")	0.22ms (kernel driver + RPC)			0.27ms

Table 2: **PhotoBooth Latency.** The total roundtrip time for sending an event from the PhotoBooth Web application through a framework and driver to a USB flash device. We compare a JavaScript driver and framework, a NaCl driver and framework, and a native desktop "application" using a Linux kernel driver.

quire additional FAT16 operations to update file system metadata. As before, the JavaScript driver and framework are the slowest, while NaCl achieves closer performance to the in-kernel driver and framework.

As a final test, we measured the frame rate at which PhotoBooth could render a live video stream for the JavaScript and NaCl versions of the driver and framework. The NaCl version could render at the full rate of 30 frames per second, but inefficient, byte-level operations to process URBs and video frames in the driver became a bottleneck of the JavaScript version, achieving only 14 frames per second.

5.1.3 Performance summary

Unsurprisingly, JavaScript and the use of user-level drivers introduce significant performance overhead relative to in-kernel, native device drivers. However, we believe that JavaScript and NaCl drivers are sufficiently fast to be practical for many USB device classes, including storage devices and video cameras, in spite of the fact that we have not attempted to optimize them, or our user-mode driver framework, in any significant way.

5.2 Security Implications

There are serious security implications to exposing local USB devices to Web programs using Maverick. In this section, we define Maverick's threat model and use it to describe the possible attacks against Maverick. For each attack, we discuss how Maverick defends against the attack and identify where our current security mechanisms are lacking.

5.2.1 Threat Model

Our threat model is similar to that assumed by modern browsers when handling untrusted extensions [1]. We assume that the browser kernel, the browser instance sandboxes, and the Maverick components inside the host OS are correctly implemented and vulnerability-free. Thus, Web drivers, frameworks, and application instances can

	20KB file create+write		20KB file read	
	throughput	CPU util.	throughput	CPU util.
JavaScript Web driver	11.2 files/s	27%	28.0 files/s	24%
NaCl Web driver	44.1 files/s	18%	134.4 files/s	18%
Linux kernel driver	65.5 files/s	0.4%	509 files/s	1.1%

Table 3: **File Throughput.** The throughput and CPU utilization of two file benchmark applications, 20KB file creates+writes and 20KB file reads, in three cases.

only be directly attacked through Maverick's IPC channels; we assume that attackers cannot directly access or modify DOM objects in Web instances, or attack Web instances by corrupting the browser kernel.

5.2.2 Attacks, Defenses, and Limitations

Given this threat model, Maverick is vulnerable to two broad classes of attacks: (1) attacks aimed at compromising *trusted* Web instances that are benign but buggy, and (2) attacks aimed at tricking Maverick into running *malicious* Web instances.

Benign-But-Buggy Web Instances. This class of attacks exploits bugs in trusted Web instances to expose a user's local devices to an attacker. Local devices often contain highly sensitive user information that an attacker might like to access, delete, modify, or corrupt. Examples include data stored on USB storage devices, environmental information obtained via audio, video, and GPS input devices, and sensitive information transmitted to printers or other peripheral devices.

Because trusted frameworks and device drivers in Maverick are Web programs, buggy implementations can be vulnerable to conventional Web-based attack techniques such as cross-site scripting, cross-site request forgery, and framing attacks. Additionally, if trusted Web instances are served over an insecure channel, attackers could mount man-in-the-middle attacks to eavesdrop on and hijack local devices.

Maverick does not provide any additional mechanisms for defending against such attacks beyond those already provided by browsers, nor does it make mounting these attacks any easier. Instead, Maverick elevates the consequences of writing buggy Web programs. Web developers will need to adopt best practices for eliminating common vulnerabilities, and ensure that Web instances are transmitted only through SSL-protected channels.

Maverick's domain-driven naming, binding, and authorization policy described in Section 3.3, in which a user configures their browser to trust a domain provider's choice of frameworks and drivers, makes it so that the average user need only trust a small number of well-known Web instance providers such as Google or Microsoft. Of course, power users that opt-out of this default setting and choose to adopt the more flexible application-driven policy will face an increased burden in vetting the quality of Web instances.

Malicious Web Instances. This second class of attacks relies on an attacker being able to run a malicious Web driver or framework in the user's browser. If successful, the consequences could be serious: because Maverick exposes the USB subsystem to Web drivers, a malicious Web driver can not only control the associated local device, but can inject arbitrary device commands into the USB subsystem. This could allow the attacker to introduce malware onto the device, or exploit bugs in the USB software stack that could give the attacker unfettered access to the user's system.

Maverick's security mechanisms are designed primarily to prevent attackers from running malicious Web instances in the first place. To try to trick Maverick into running a malicious Web instance, attackers might leverage existing network-based and social engineering attacks.

Example network-based attacks include DNS rebinding and DNS cache-poisoning which redirect valid DNS mappings in the browser to malicious URLs. Maverick could defend against these attacks by using stronger naming mechanisms such as secure DNS or certificate-based naming. Maverick also protects against social engineering attacks by relying on the domain-driven naming, binding, and authorization policy to prevent users from authorizing malicious drivers or frameworks.

If a malicious Web instance somehow does manage to run in the browser despite these safeguards, Maverick currently has limited ability to protect the local system. At best, Maverick's IPC channels serve to help mitigate the damage an attacker can do by limiting the compromised device driver's access to a single USB device. In general, once an attacker has the ability to run a malicious Web driver or framework, we consider the user's computing environment to be compromised.

5.3 Experiences with Maverick

Our experience with programming Maverick drivers and frameworks has been positive. JavaScript and NaCl drivers and frameworks enjoy Maverick's clean event model. Programmers do not need to worry about kernel synchronization, pitfalls of interrupt-context code, or the vagaries of kernel memory allocation. If a driver is buggy, that browser instance will crash, but other browser instances and the host OS continue to execute. NaCl makes it easy to port existing C kernel drivers and frameworks into Maverick. While JavaScript is slower and not yet ideally suited for manipulating raw, binary data, JavaScript code is dynamic and flexible, so managing complexity like callback function pointers is straightforward. Overall, we found that for the majority of devices we considered, the advantages of programming in higher-level languages and abstractions far outweighed any performance limitations.

To showcase the power and flexibility of Maverick, we built two additional Web applications. The first, called SquirtLinux, makes it trivial to install Linux onto a USB flash drive. Linux thumbdrives are useful for rescuing data from a damaged system or carrying a portable, secure boot environment. SquirtLinux consists of combined Web application and framework instance that exploits our existing JavaScript-based USB mass storage driver. SquirtLinux communicates with a Web server, having it prepare a "Puppy Linux" USB installation image on behalf of the user; it then uses AJAX to pipeline downloading blocks of the image with writing it to an attached drive. As a result, the user simply needs to find a Maverick browser, attach their flash drive, browse to the SquirtLinux service, and press a single button to manufacture a bootable thumbdrive.

Our second application, WebAmbient, uses a Delcomm USB LED indicator to build an ambient display. The LED supports a custom USB protocol that lets it be programmed to emit any color as a combination of red, green, and blue light. We wrote a JavaScript Web driver and Web framework for it that exposes a high-level color toggling interface to applications. Next, we wrote a Web mashup that fetches real-time stock prices, making the LED grow redder as price drops, greener as it increases, and blue if it is flat. Maverick's flexibility made it simple for us to support the custom USB protocol and expose high level functions without requiring any change to the host OS, browser, or HTML standards.

6 Related Work

Maverick builds upon architectural features and techniques explored by prior work. We discuss related work below.

6.1 User-Level Drivers and Frameworks

Maverick moves device driver and framework code out of the OS and into (user-level) Web applications. Our approach of running drivers and frameworks as Web applications is new, but the general approach of deconstructing a monolithic OS and moving its components to the user-level is well studied. Mach [6] and L3 [16] explored microkernels, small OS kernels that provide basic hardware abstraction and rely on user-level servers to implement major OS subsystems. Maverick also shares features with exokernels [14], which allocate and protect hardware resources within a small, trusted kernel and delegate the higher-level abstractions to user-level programs.

Prior work has explored OS support for user-level drivers. Decaf [22] and Microdrivers [7] use static program analysis and code annotations to automatically partition kernel drivers into user-space and kernel components, leaving just the performance critical components (like I/O) in the kernel. Other user-level driver frameworks have been implemented on top of Linux [15, 5] and Windows [17].

Maverick's virtualization of the USB hub is perhaps closest to the open source libusb [4] and Javax.USB [11] projects, which expose the Linux USB core API to user-level code. Like libusb and Javax.USB, Maverick exposes familiar USB API functions, but unlike libusb and Javax.USB, Maverick Web drivers are untrusted and dynamically located, downloaded, and executed.

6.2 Browser Architecture

Maverick inherits many of the safety and reliability benefits of the Chrome browser. Chrome maps Web applications into OS processes, providing better isolation and resource management [20, 19, 21]. Chrome's process model allows Maverick instances to run in parallel on multicore systems, and its sandbox isolates untrusted code.

Other research browsers have similar properties. For example, Gazelle [23] isolates Web applications into processes using the same origin policy (SOP), placing mashup content from separate Web origins in different trust domains. Gazelle plugins execute in their own processes, and the browser kernel protects updates to the display. OP decomposed the browser architecture into multiple trusted components, but it did not provide full compatibility with existing sites [10]. SubOS provided a process abstraction for Web applications, but did not explore in detail a process model or the interactions between processes [13]. Tahoma provided safe and flexible isolation between Web applications by embedding each in a Xen virtual machine [2]. Tahoma did not consider the problem of exposing local devices to Web applications, but rather provides them with a limited set of standard virtual devices.

6.3 Web Application Access to Devices

ServiceOS's browser architecture is designed to isolate Web applications and allow them to directly monitor and manage device resources [18]. As with Chrome OS [8], ServiceOS envisions itself as a browser OS. However, ServiceOS exposes devices to Web applications by including DOM objects and system calls in the browser kernel on a device-per-device basis. This solution is clean and simple, but suffers from the same drawbacks as adding new HTML tags to the HTML specification: new device support requires modifications to the browser. In contrast, Maverick lets Web developers add support for new devices and functionality *without* requiring browser vendor or OS cooperation.

The Javax.USB [11] project allows signed Java applets to access USB devices. This work is complimentary to the Web drivers piece of Maverick. However, Javax.USB requires use of the heavy-weight (and potentially unsafe) Java browser plugin, and provides no explicit support for safe IPC channels for direct framework and application communication with drivers.

6.4 Client-Side Code Sandboxes

Maverick uses Native Client [24] to provide a safe, client-side sandbox for executing x86 Web drivers and frameworks built with languages like C and C++. These languages handle certain aspects of driver development such as byte-level data manipulation more efficiently than JavaScript, and supporting them in Maverick makes porting existing kernel drivers much easier. Similar technologies to Native Client exist, most notably Xax [3]. Other client-side sandboxes include Flash, Java, Silverlight, and ActiveX. Although our current implementation focuses only on JavaScript and Native Client, Maverick could support these sandboxes in the future.

7 Conclusions

The Maverick browser gives Web applications safe access to local USB devices, and permits programmers to implement USB device drivers and frameworks using standard Web programming technologies like JavaScript and native client (NaCl). With Maverick, Web drivers and Web frameworks are downloaded dynamically from servers and safely executed by the Maverick browser kernel, allowing new drivers and frameworks to be implemented, distributed, and executed as conveniently as Web applications.

The main challenges faced by Maverick are safety and performance. Our prototype system, built by extending the Chrome browser and Linux kernel, relies on existing sandboxes to isolate Web drivers and frameworks from each other, the Maverick browser, and the host operating system and applications. As well, our system is architected to be flexible in supporting a range of policies and trust models for authorizing Maverick driver, framework, and application access to specific USB devices, and for resolving which drivers and frameworks ought to be downloaded and executed to satisfy application dependencies.

We prototyped several JavaScript and NaCl drivers and frameworks, and evaluated them using microbenchmarks and application workloads. While our measurements confirm that JavaScript Web drivers and frameworks suffer from much higher latency and lower throughput than NaCl or Linux equivalents, we also showed that they perform sufficiently well to drive interesting USB devices, including Webcams and USB flash storage devices. Finally, we described two Web applications that showcase the flexibility and power of the Maverick approach.

Acknowledgments

We thank Paul Gauthier, Charlie Reis, Roxana Geambasu, and Ed Lazowska for their feedback and help. We also thank our anonymous reviewers, and our shepherd Jon Howell, for their guidance. This work was supported by NSF grants CNS-0627367 and CNS-1016477, the Torode Family Career Development Professorship, and gifts from Intel Corporation and Nortel Networks.

References

[1] Adam Barth, Adrienne Porter Felt, Prateek Saxena, and Aaron Boodman. Protecting browsers from extension vulnerabilities. In *Proceedings of the 17th Network and Distributed System Security Symposium (NDSS 2010)*, San Diego, CA, February 2010.

[2] Richard S. Cox, Steven D. Gribble, Henry M. Levy, and Jacob Gorm Hansen. A safety-oriented platform for Web applications. In *Proceedings of the 2006 IEEE Symposium on Security and Privacy*, Washington, DC, May 2006.

[3] John R. Douceur, Jeremy Elson, Jon Howell, and Jacob R. Lorch. Leveraging legacy code to deploy desktop applications on the Web. In *Proceedings of OSDI 2008*, San Diego, CA, December 2008.

[4] Daniel Drake and Peter Stuge. libusb. `http://www.libusb.org/`.

[5] J. Elson. FUSD: A Linux framework for user-space devices. `http://www.circlemud.org/~jelson/software/fusd/`.

[6] Alessandro Forin, David Golub, and Brian Bershad. An I/O system for Mach 3.0. In *Proceedings of the USENIX Mach Symposium*, Monterey, CA, November 1991.

[7] Vinod Ganapathy, Matthew J. Renzelmann an Arini Balakrishnan, Michael M. Swift, and Somesh Jha. The design and implementation of Microdrivers. In *Proceedings of ASPLOS 2008*, Seattle, WA, March 2008.

[8] Google. Chromium OS. `http://www.chromium.org/chromium-os`.

[9] Google. Protocol buffers. `http://code.google.com/p/protobuf/`.

[10] Chris Grier, Shuo Tang, and Samuel T. King. Secure Web browsing with the OP Web browser. In *Proceedings of the 2008 IEEE Symposium on Security and Privacy*, Washington, DC, May 2008.

[11] IBM. Javax.USB. `http://www.javax-usb.org/`.

[12] IETF. RFC 2397: The data URL scheme. `http://tools.ietf.org/html/rfc2397`.

[13] Sotiris Ioannidis and Steven M. Bellovin. Building a secure Web browser. In *Proceedings of the FREENIX of the USENIX ATC*, Boston, MA, October 2001.

[14] M. Frans Kaashoek, Dawson R. Engler, Gregory R. Ganger, Hctor M. Briceo, Russell Hunt, David Mazires, Thomas Pinckney, Robert Grimm, John Jannotti, and Kenneth Mackenzie. Application performance and flexibility on Exokernel systems. In *Proceedings of SOSP 1997*, Saint Malo, France, September 1997.

[15] Ben Leslie, Peter Chubb, Nicholas Fitzroy-dale, Stefan Götz, Charles Gray, Luke Macpherson, Daniel Potts, Yueting Shen, Kevin Elphinstone, and Gernot Heiser. User-level device drivers: Achieved performance. *Journal of Computer Science and Technology*, 20(5), 2005.

[16] Jochen Liedtke, Ulrich Bartling, Uwe Beyer, Dietmar Heinrichs, Rudolf Ruland, and Gyula Szalay. Two years of experience with a μ-kernel based OS. *SIGOPS Operating System Review*, 25(2), 1991.

[17] Microsoft. Architecture of the user mode driver framework. `http://www.microsoft.com/whdc/driver/wdf/UMDF-arch.mspx`, February 2007.

[18] Alexander Moshchuk and Helen J. Wang. Resource management for Web applications in ServiceOS. In *MSR-TR-2010-56*, Redmond, WA, May 2010.

[19] Charles Reis, Brian Bershad, Steven D. Gribble, and Henry M. Levy. Using processes to improve the reliability of browser-based applications. In *University of Washington Technical Report UW-CSE-2007-12-01*, Seattle, WA, 2007.

[20] Charles Reis and Steven D. Gribble. Isolating Web programs in modern browser architectures. In *Proceedings of EuroSys 2009*, Nuremberg, Germany, April 2009.

[21] Charles Reis, Steven D. Gribble, and Henry M. Levy. Architectural principles for safe Web programs. In *Proceedings of HotNets 2007*, Atlanta, GA, November 2007.

[22] Matthew J. Renzelmann and Michael M. Swift. Decaf: Moving device drivers to a modern language. In *Proceedings of the USENIX ATC*, San Diego, CA, June 2009.

[23] Helen J. Wang, Chris Grier, Alexander Moshchuk, Samuel T. King, Piali Choudhury, and Herman Venter. The multi-principal OS construction of the Gazelle Web browser. In *Proceedings of USENIX Security 2009*, Montreal, Canada, August 2009.

[24] Bennet Yee, David Sehr, Gregory Dardyk, J. Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. Native Client: A sandbox for portable, untrusted x86 native code. In *Proceedings of the 30th IEEE Symposium on Security and Privacy*, Oakland, CA, May 2009.