Flashproxy: Transparently Enabling Rich Web Content via Remote Execution

Alexander Moshchuk, Steven D. Gribble, and Henry M. Levy
Department of Computer Science & Engineering
University of Washington, Seattle, WA, USA 98195
{anm, gribble, levy}@cs.washington.edu

ABSTRACT

It is now common for Web sites to use active Web content, such as Flash, Silverlight, or Java applets, to support rich, interactive applications. For many mobile devices, however, supporting active content is problematic. First, the physical resource requirements of the browser plug-ins that execute active content may exceed the capabilities of the device. Second, plug-ins are simply not available for many devices. Finally, active code and the plug-ins that execute it often contain security flaws, potentially exposing a user's device or private data to harm.

This paper explores a proxy-based approach for transparently supporting active Web content on mobile devices. Our approach uses a proxy to splice active content out of Web pages and replace it with an AJAX-based remote display component. The spliced active content executes within a remote sandbox on the proxy, but it appears embedded in the Web page on the mobile device's browser.

To demonstrate the viability of this approach, we have designed, implemented, and evaluated Flashproxy. By using Flashproxy, any mobile Web browser that supports JavaScript transparently inherits the ability to access sites that contain Flash programs. The major challenge in Flashproxy is in trapping and handling interactions between the Flash program and its execution environment, including browser interactions. Flashproxy uses binary rewriting of Flash bytecode to interpose on such interactions, redirecting them through a JavaScript-based RPC layer to the user's browser. Our evaluation of Flashproxy shows that it is transparent, performant, and compatible with nearly all Flash programs that we examined.

Categories and Subject Descriptors

H.3.4 [**Systems and Software**]: Distributed systems; H.3.5 [**Online Information Systems**]: Web-based services

General Terms

Design, Performance

Keywords

Web browsers, Flash, proxy, binary rewriting, active Web content

MobiSys'08, June 17–20, 2008, Breckenridge, Colorado, USA.

1. INTRODUCTION

Today's Web sites often use active content such as Flash, Java, or Silverlight to provide their users with rich, interactive applications. In many environments, including mobile device Web browsing, browser support for active content is problematic. Active content types require complex plug-ins to run, yet on many device classes these plug-ins are not yet available. Even when they are available, installing and running browser plug-ins can be undesirable, as they create security vulnerabilities, require significant compute and memory resources, and create administrative burden on users to keep them patched and up to date.

As a concrete example, recent mobile devices such as Windows Mobile PDAs and the Apple iPhone contain feature rich browsers. However, Windows Mobile devices only support Flash Lite, a trimmed down version of Flash, and the iPhone does not yet have any Flash support. Flash has also had its share of security flaws, including critical vulnerabilities that can be exploited by remote Web sites [26] or that make the browser susceptible to denial of service attacks [25].

This paper demonstrates that it is possible for a mobile-device-based browser to access Web sites that contain active content, even if the browser does not have built-in support for it. Our approach relies on a proxy to intercept and modify the Web page, splicing out any active content it finds and replacing it with an AJAX-based [12] remote display Web component. The active content executes within a secure sandbox on the proxy, but it is remotely displayed in the correct location within the mobile browser's Web page.

This approach has many benefits. First, it is *transparent*; any Web browser that supports JavaScript and that has sufficient bandwidth to the proxy gains support for the active content, without requiring any new software on the device. Second, it is more *secure*; vulnerabilities in the plug-ins that support the active content are isolated from the user, her browser, and her device. Third, it is more *manageable*; if any software updates or security patches need to be applied to the plug-in, they can be done at the proxy without involving users.

To investigate the viability of our approach, we designed, implemented, and evaluated Flashproxy, a system that provides support for Flash content in JavaScript-enabled browsers. Flashproxy was architected to be extensible to additional content types, including Silverlight and Java. However, we implemented Flash support first, since it is far more common in today's Web. With Flashproxy, a Web browser no longer needs a Flash plug-in; instead, Flashproxy splices out the Flash content, replacing it with only 10KB of JavaScript code.

Flashproxy posed many challenges, primary of which is correctly handling interactions between the Flash program and its execution environment. A Flash program can communicate with a re-

mote Web server, trigger the download of an object into the user's file system, cause the browser to navigate to a new page, or directly invoke JavaScript functions defined within its enclosing page. To preserve the functionality of the Web page, Flashproxy must intercept these interactions on the remote proxy and either handle them directly or forward them to the user's browser. Our implementation relies on binary rewriting of Flash bytecode to interpose on the execution of the Flash program, and an RPC layer implemented in JavaScript to forward relevant function invocations between the remotely executing Flash program and the mobile device's browser.

We evaluated Flashproxy using microbenchmarks, manual examination of real Flash applications, and automated analysis of Flash programs found using a Web crawler. Our evaluation shows that nearly all Flash programs execute correctly through Flashproxy, including on mobile devices such as the iPod Touch. Though we have not yet tried to optimize the performance of the remote display or RPC components of Flashproxy, we found it performs well for many commonly encountered Flash application classes, including Flash-based navigation structures, advertisements, and YouTube-style embedded Flash movies. Overall, our experience demonstrates the viability of using the Flashproxy architecture for executing active content on behalf of mobile devices.

2. AN OVERVIEW OF FLASH

To provide context for the rest of this paper, this section gives a brief technical overview of Flash. We discuss the structure of Flash applications, their common uses, and their execution environment.

A Flash application, colloquially referred to as a "Flash movie," is a program that contains a blend of vector and raster graphics, audio and video components, and bytecode that is typically compiled from a scripting language called ActionScript. In the context of the Web, Flash programs are active objects that are embedded within Web pages, similar to Java applets or Silverlight applications. Flash has gained widespread adoption in recent years, and it is being used to add many kinds of interactive content to Web pages.

Flash programs are developed using integrated development environments (IDEs) that are geared more towards supporting Web and multimedia development than traditional software engineering tasks. An IDE compiles a Flash program into a `.swf`-formatted file that contains the media objects, user interface elements, and bytecode used by the program. A Flash virtual machine parses and executes the `.swf` file; Web browsers rely on "Flash Player" plug-ins to implement this virtual machine.

In addition to the basic virtual machine architecture, a Flash Player also defines built-in classes that Flash bytecode, and therefore ActionScript programs, can invoke. These built-in classes (named `flash.*`) play a similar role to Java's class library. Some `flash.*` classes facilitate the manipulation of graphics, user interface elements, and events. Others allow bytecode to take advantage of underlying operating system functions, including raw network, file, and HTTP operations. As well, the `flash.*` classes allow a Flash program to interact with JavaScript functions defined by the Web page in which it is embedded, or to ask the browser to navigate to a new Web page. Figure 1 shows a high-level architectural view of a Flash program running within a browser.

Flash and ActionScript have evolved over time. In this paper, we focus only on Flash versions 1–8, associated with the ActionScript language version 2 or below. Adobe recently released a redesigned Flash virtual machine, called AVM2, along with ActionScript 3.0 and Flash Player 9. From our preliminary investigation, the abstractions and mechanisms we have designed in Flashproxy should be applicable to AVM2 programs, though to handle them we would need to implement a new bytecode parser and binary rewriter. Fortunately, our evaluation in Section 5 shows that the vast majority of Flash programs on the Web are not AVM2-based.

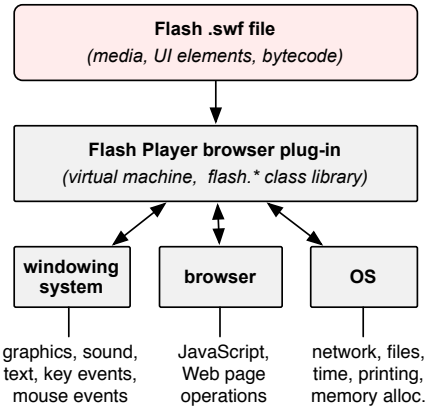


Figure 1: Flash architecture. A Flash program is contained within a .swf file, and is executed by a virtual machine implemented by the Flash Player browser plug-in. The Flash Player also defines a set of classes that allow the Flash program to interact with the host OS, its windowing system, and the Web page and browser in which the Flash program runs.

2.1 Flash bytecode

Flash bytecode consists of a sequence of instructions; an instruction is commonly referred to as an "action." The Flash specification defines what instructions are understood by the Flash virtual machine and the opcodes into which the instructions are encoded. The Flash virtual machine is stack-oriented; an instruction can specify some arguments directly in its encoding, but it will more generally pop its arguments from the stack and push a result back onto it. The Flash virtual machine also has a simple notion of a program counter (PC). The value of the PC is defined to be the address of the instruction that follows the instruction currently being executed. The instruction address space is byte oriented, but encoded instructions may require several bytes. The Flash verifier and virtual machine only permit programs to transfer control to addresses associated with the beginning of an instruction opcode.

Flash defines instructions for arithmetic, logic, string manipulation, variable manipulation, control flow, type conversion, and exception handling. More recent versions of Flash include object-oriented instructions, including operators for class and method definition, lookup, and invocation. Flash also has a number of higher-level, CISC-like instructions, including ones that manipulate the timeline of a Flash program or retrieve media objects over HTTP. Many of these high-level instructions have become deprecated in favor of functionally equivalent `flash.*` classes and methods implemented by the Flash Player.

As we will describe, Flashproxy relies on binary rewriting to interpose on certain functions that a Flash program might invoke. Flash provides two basic mechanisms for function invocation. First, a program can issue one of the CISC-like instructions that the virtual machine defines; for example, the `ActionGetURL2` instruction lets a program retrieve Web data into a local variable or command the Web browser to navigate to a new page. Second, a program can invoke a function or method; some functions and methods are defined by the `flash.*` classes, while others can be defined by the program itself. To call a function or method, a program pushes

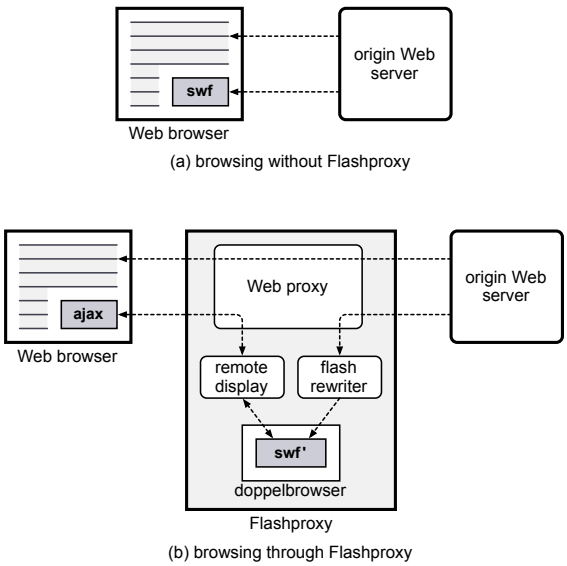


Figure 2: High-level Flashproxy architecture. (a) A browser directly fetching a Web page with an embedded Flash program. (b) A browser using Flashproxy to fetch the same Web page; Flashproxy rewrites the Web page, replacing the Flash program with a remote display AJAX component. The Flash program executes within a "doppelbrowser" – a full Web browser and Flash Player executing in Flashproxy.

onto the stack either the String name of the callee or a reference to the callee, as well as any arguments, and then invokes either the `ActionCallFunction` or `ActionCallMethod` instruction. The virtual machine will pop the appropriate values off the stack and transfer control to the callee.

3. ARCHITECTURE

In Figure 2, we present a high-level, conceptual illustration of the Flashproxy architecture. To use Flashproxy, a user simply configures her browser to proxy all HTTP requests through Flashproxy; no software needs to be installed on the client.

If Flashproxy detects that an embedded Flash program exists on a requested Web page, it rewrites the Web page to splice out the embedded program, replacing it instead with our AJAX-based remote display Web component. This component relays user input to Flashproxy and downloads its graphical display from Flashproxy.

On the proxy-side, Flashproxy passes the spliced-out Flash program through a binary rewriting component to insert interposition hooks on functions of interest. The rewritten Flash program is then embedded inside a minimal Web page and rendered by a *doppelbrowser* – a fully-functioning Web browser executing within Flashproxy. A remote display server on Flashproxy relays the input and output of the Flash program to the AJAX component on the client's browser. If a Web page contains multiple Flash programs, FlashProxy will run multiple doppelbrowser instances, one per program.

Figure 2 is deliberately simplified; it elides additional components that are necessary to make Flashproxy work. To keep Flashproxy transparent, interactions between Flash programs and their execution environment must be interposed upon and relayed to the user's Web browser if necessary. In the rest of this section, we identify the resulting complications and describe the architectural components we constructed to handle them.

3.1 Interactions between Flash and remote hosts

Flash code can interact with remote hosts, either by downloading data over HTTP or by opening raw network sockets to remote ports. As a security measure, the Flash player enforces a *same origin policy* on remote communications. As a default, this policy prevents a Flash program from communicating with any host other than the one from which it was downloaded. Therefore, to preserve the ability for Flash to communicate, Flashproxy must make the doppelbrowser believe that the Web page and rewritten Flash program it is executing were downloaded from the true origin Web site. As well, Flashproxy must preserve the pathname of the embedded program, so that any relative URLs generated by it cause the doppelbrowser or Flash Player to issue HTTP requests with correct absolute pathnames.

To accomplish this, we enhanced the Web proxy to look for two specific URLs: that of the Web page embedding the Flash object, and that of the Flash object itself. If the Web proxy sees either of these URLs coming from the doppelbrowser, it redirects them to an internal Web server we run on Flashproxy. All other network connections (HTTP or raw sockets) emanating from the doppelbrowser are permitted to pass unmodified; we rely on the doppelbrowser to enforce the same origin policy over the rewritten Flash program, and assume that such communication is safe. When the doppelbrowser is initially launched, Flashproxy directs it to load the embedding Web page; we seed the internal Web browser with a simple page containing only the rewritten Flash object.

Flash Player provides classes that allow a program to dynamically load additional Flash code over HTTP and execute it as part of the same program. Our Web proxy looks for any such dynamic load requests coming from the doppelbrowser, and uses the same mechanisms as described above to rewrite this new code and serve it from the internal Web server.

3.2 Interactions between Flash and JavaScript

Some Flash Player classes let a Flash program invoke JavaScript functions defined on the Web page in which the Flash program is embedded (Figure 3a). When invoked, the Flash virtual machine transfers the logical thread of execution to the JavaScript interpreter running in the Web browser, and resumes once the function has finished. Because the JavaScript functions can have side-effects visible to the user, and because the functions might depend on state only present on the user's browser, we cannot execute these JavaScript functions on the doppelbrowser. Instead, we must execute them on the user's browser, transferring any arguments and return value between the calling Flash program and the remote JavaScript function.

To do this, we implemented a JavaScript RPC layer, as shown in Figure 3b. The binary rewriter modifies any `ExternalInterface` calls in the Flash program to invoke a stub function in our doppelbrowser's Web page. The stub function marshals the function name to be invoked and its arguments into a string, and uses XMLRPC to pass this string to an *RPC nexus* process running in FlashProxy.

FlashProxy inserts a corresponding skeleton JavaScript function into the user's Web page before transferring it to the user's browser. This skeleton function uses XMLRPC to pull new RPC invocations from the RPC nexus. After receiving a new invocation, the skeleton unmarshals it and invokes the appropriate JavaScript function. The return path to the Flash program in the doppelbrowser proceeds similarly.

The RPC nexus is needed as JavaScript cannot receive incoming network connections, but only call out to remote Web servers. The

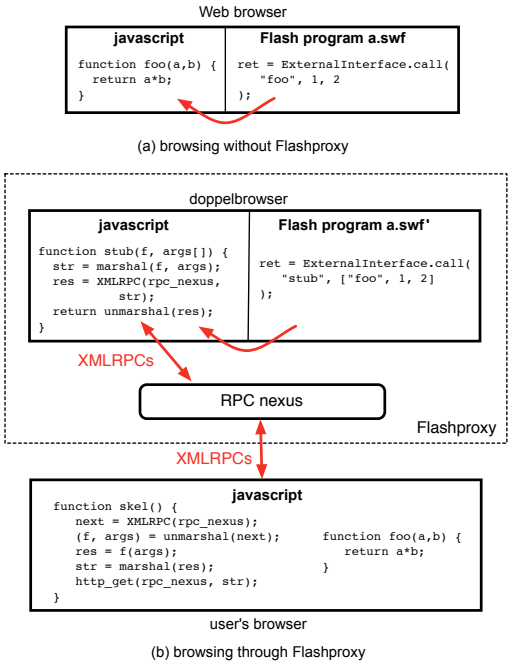


Figure 3: JavaScript invocation. (a) A Flash program can use ExternalInterface to call JavaScript functions. (b) FlashProxy rewrites Flash programs to call instead a stub JavaScript function on the doppelganger web page. This stub marshals the request and relays it through an "RPC nexus" process to a corresponding skeleton function that FlashProxy inserted on the user's Web page. The skeleton invokes the appropriate JavaScript function.

nexus therefore acts as an intermediary that connects the JavaScript stub and skeleton together. Though this design adds extra overhead, our evaluation shows that `ExternalInterface` calls are rare, and hence the overhead will not affect the common case performance of a Flash program. Using the nexus allows us to implement the JavaScript RPC layer on the user's browser without requiring that the user install any new software; the skeleton is simply JavaScript code inserted by the proxy.

The `ExternalInterface` class also allows Flash programs to register functions that JavaScript running in a browser can invoke. The mechanism we use to handle this is similar to that described above, but it proceeds in the opposite direction. A stub function on the user's browser marshals requests and relays them through the RPC nexus to a JavaScript skeleton on the doppelbrowser. This skeleton invokes the function within the Flash player. The only additional complication is that the function registration event generated by the Flash program needs to be relayed to the user's browser, so that an appropriate stub function can be created there. We interpose on Flash `ExternalInterface` registration events using binary rewriting, and use the JavaScript RPC layer to pass this event to skeletons running on the user's browser.

3.3 Interactions between Flash and the browser

Several Flash instructions and classes, including `fscommand`, `getURL`, `getURL2`, and `movieclip.getURL`, allow a Flash program to interact with the browser. For example, `fscommand` can be used to change the Flash program to full-screen mode, and `getURL` can cause the browser to open a specified URL in a new page.

Our binary rewriter identifies these calls, and translates them into `ExternalInterface` calls to a JavaScript stub function on the doppelbrowser Web page. From there, the calls are marshaled and relayed to the user's browser using the same RPC layer described above. The proxy inserts a skeleton JavaScript function into the user's Web page that unmarshals these calls and causes the appropriate browser side-effect to happen.

Flash also provides `flash.net.LocalConnection`, a class that allows two Flash programs embedded within the same page to directly communicate with each other. We have not yet implemented support for this class, but doing so should be a simple matter of using our binary rewriter to redirect these calls to JavaScript stubs on the respective browsers, and to use the JavaScript RPC layer to relay data between the stubs.

3.4 Detecting embedded Flash content

Determining if a Web page contains a Flash program is surprisingly intricate. The simplest way to embed Flash is to use `<embed>` HTML tag with the appropriate type, height, width, and src attributes. If a page uses this method, our Web proxy can detect and splice out these embed tags, replacing them with our AJAX element instead. At that time, the proxy also downloads the referred Flash script and passes it to the rewriter.

The `<embed>` tag is not endorsed by the W3C; instead, they mandate the use of the `<object>` element. Unfortunately, Microsoft's IE browser interprets `<object>` elements slightly differently than most other browsers, leading Web sites that use this tag to actually embed two nested versions of it, one for IE and one for other browsers. Consequently, scanning HTML to look for Flash elements embedded with this tag is tricky.

To make matters worse, many Web sites use JavaScript to test whether the client's browser supports Flash; if so, it is common for the JavaScript code to dynamically rewrite the HTML (using `document.write()` or `innerHTML={...}`) to inject `<embed>` or `<object>` tags into the page. A popular script package called SWFObject encapsulates all of this complexity.

It is not possible for the Web proxy to detect Flash on pages that use these dynamic techniques, since the HTML that ultimately embeds the object is generated as a side-effect of running client-side JavaScript. Instead, we use the proxy to inject our own JavaScript-based detection code into all transferred pages. Our code periodically inspects the DOM of the page to look for embedded objects associated with Flash. If one is found, our code replaces the object with an AJAX remote display widget and uses the JavaScript RPC layer to notify Flashproxy, which then downloads, rewrites, and executes the Flash program within a doppelbrowser.

As a final intricacy, scripts such as SWFObject probe the list of installed browser plug-ins to test for Flash support. Our injected JavaScript code fools these scripts into believing that a Flash plug-in exists, causing the `<embed>` or `<object>` tag to be inserted and the rest of the page to believe the Flash program is running.

3.5 Summary

This section presented a simplified FlashProxy architecture, and then described a set of additional components and interfaces that we required to interpose upon and transparently handle interactions between a Flash program and its execution environment. Figure 4 illustrates the resulting architecture. Through a combination of binary rewriting to interpose on Flash functions, JavaScript stubs and skeletons inserted on the user's and doppelbrowser's Web page, and an RPC nexus to intermediate between them, we were able to catch and redirect these interactions in a way that is transparent to the user's device and the origin Web server.

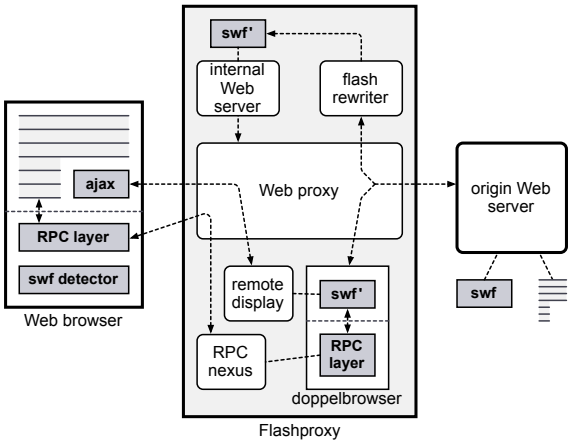


Figure 4: Detailed Flashproxy architecture. This diagram illustrates the additional components we needed to add to Flashproxy to handle a Flash program's external interactions in a way that is transparent to both the user's browser and the origin Web server.

4. IMPLEMENTATION

In this section of the paper, we provide additional implementation details of three FlashProxy components: the Flash binary rewriter, the JavaScript-based RPC layer, and the AJAX-based remote display Web widget. As well, we provide some details about the JavaScript code that Flashproxy injects into the user's browser.

4.1 Binary rewriter

The binary rewriter component must inspect and modify the bytecode embedded in a Flash program, and then update the `.swf` file parameters and headers to account for any changes, including the length of bytecode stored within it. Instead of implementing our own bytecode parser, we used an open-source tool called Flasm [14] to disassemble the bytecode in a Flash program into a textual representation. The binary rewriter operates on this textual representation and then uses Flasm's assembler to update the bytecode within the `.swf` file.

The rewriter searches for specific bytecode instructions and replaces them with an alternate sequence of instructions. For example, an `ActionGetURL` instruction can take a URL embedded directly in its opcode and redirect the client's Web browser to this URL. We rewrite this instruction so that the program instead performs an `ExternalInterface` call to pass the URL to the RPC layer in the doppelbrowser. The RPC layer forwards this URL to the RPC nexus, which relays it to the RPC layer in the client's browser. To replace `ActionGetURL` with this `ExternalInterface` call, the rewriter must inject several new instructions that manipulate the execution stack to properly construct arguments for the `ExternalInterface` call and then invoke the call using an `ActionCallMethod` instruction.

As another example, we rewrite all instructions that invoke a function call, specifically `ActionCallFunction` and `ActionCallMethod`. We change them to first check their call targets against a list of functions we need to interpose on, such as `ExternalInterface.call` or `ExternalInterface.addCallback`. Doing this at the bytecode level is intricate; we must insert instructions that load the address of each function we want to test against into the stack and to compare them to the actual call target. If there is a match, we take a conditional branch to code that makes an `ExternalInterface` call to the doppelbrowser's RPC layer, passing the original function's arguments as well as the matched function's name as an extra argument. If there is no match, we invoke the original call, being careful to preserve its original arguments on the stack. To compare functions, we use pointers to functions rather than names to account for cases where a user creates custom variables pointing to functions.

Calling a function from Flash's class library involves several lookups. For example, the `ExternalInterface.addCallback` function is located in the `flash.external` package. To call it, the code must first load the `flash` class, then look up three of its members in turn to get a function pointer. We need to perform these lookups to load each function that we must interpose on, for every call instruction we encounter. As an optimization, we avoid repeating these lookups by caching their results in global variables, which we then use for subsequent comparisons.

4.2 JavaScript-based RPC layer

Flashproxy's RPC layer is responsible for providing a bi-directional communication channel between the client's browser and the doppelbrowser. Its implementation relies on JavaScript code injected into both the client's browser and the doppelbrowser. Because we must use JavaScript, we are limited by its restrictions. For example, JavaScript cannot accept an incoming network connection, and any outgoing HTTP requests must adhere to the same-origin policy. Fortunately, JavaScript has an `XmlHttpRequest` function, which can be used to make HTTP requests either synchronously or asynchronously. This function forms the basis for our RPC layer implementation.

Because JavaScript cannot accept an incoming network connection, the client's browser and the doppelbrowser cannot connect to each other directly. Instead, they communicate by connecting to the RPC nexus and relaying information through it. If the doppelbrowser wants to make an RPC call to the client's browser, it sends its request to the RPC nexus using JavaScript's `XmlHttpRequest` function in a *synchronous* mode. This will suspend any JavaScript execution on the doppelbrowser until it receives the return value for the request, which matches the synchronous semantics of a function call in Flash.

The RPC nexus receives this request and must forward it to the client's browser. However, it cannot directly connect to it; instead, the client periodically polls the RPC nexus for updates using *asynchronous* `XmlHttpRequests`. The RPC nexus can then deliver the doppelbrowser's request as a reply to one of these `XmlHttpRequests`.

This polling approach introduces a potentially large latency for delivering RPC calls, but we were able to optimize this away. When the client's browser asks the RPC nexus for an update, the RPC nexus *delays* its response until it has an RPC call to send. JavaScript imposes a 10-second timeout on asynchronous `XmlHttpRequests`, however, meaning the client's requests cannot be delayed by more than 10 seconds. To deal with the timeout, the client initiates a new request to the RPC nexus every 5 seconds, but the RPC nexus delays the response to the latest poll request only, canceling any previous outstanding requests before they cause a timeout on the client.

When the client finishes processing an incoming RPC call, it must pass a return value back to the doppelbrowser. To do this, it makes an `XmlHttpRequest` to the RPC nexus, which then wakes up the doppelbrowser's connection and sends it the return value. Our RPC scheme is bi-directional; it works exactly the same for handling the client browser's outgoing RPC calls.

To marshal arguments between Flash and JavaScript, we take

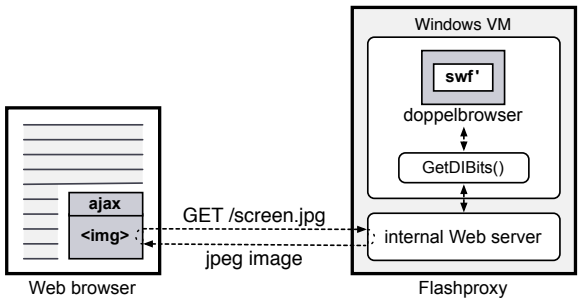


Figure 5: Remote display. Flashproxy uses the `GetDIBits` Windows system call to grab screenshots of the Flash program executing in the doppelbrowser on a Windows VM. The remote display AJAX component on the user's browser pulls screenshot updates from the Flashproxy internal Web server.

advantage of Flash's built-in marshaling provided by `ExternalInterface`. To marshal arguments between JavaScript located in the doppelbrowser and on the client, we use the JSON format and an open-source JSON library [16].

The client must be able to handle several types of RPC calls from Flash. For example, it must be able to redirect the browser to a new URL, which is necessary to support Flash's `ActionGetURL` instruction. Incoming Flash-to-JavaScript function calls amount to simple local function invocations. A more complex example is the `ExternalInterface.addCallback` call, which defines a Flash function that JavaScript can invoke in a Flash movie. To handle `addCallback`, the client's RPC layer will synthesize and inject a corresponding stub function into the document's DOM, which will enable code on the Web page to later find and invoke the Flash function. This invocation would follow the execution flow described above, but in reverse.

The RPC nexus is implemented as an extended HTTP server written in C++. It implements the request handling logic described above and maintains data structures to forward calls in both directions. Since both JavaScript and Flash's ActionScript are single-threaded, the RPC nexus needs to support at most one outstanding call from either the client's browser or the doppelbrowser, simplifying its implementation. However, the RPC nexus does handle the case of multiple Flash programs embedded within the same Web page.

One intricacy of the implementation is that `XmlHttpRequests` made by the client and the doppelbrowser are restricted by the browser's same-origin policy. The origin for RPC nexus and remote Web sites is different; thus, browsers would disallow direct communication with the RPC nexus. Fortunately, both browsers access the Web through our proxy. When we issue `XmlHttpRequests` to the RPC nexus, we prepend the URLs with the remote origin followed by a special marker that instructs the proxy to forward the request to the RPC nexus.

4.3 AJAX remote display

The AJAX remote display component runs in the client's browser and is responsible for displaying the graphical output of the Flash program and for capturing and forwarding user input, such as mouse clicks or keystrokes. We have implemented a fairly simple remote display protocol; more sophisticated protocols, discussed in section 7.3, would be applicable to Flashproxy.

To implement remote display, we periodically grab screenshots of the Flash program running in the doppelbrowser and deliver them to the client. Our doppelbrowser runs in a Windows virtual machine. We use the Windows API functions `BitBlt` and `GetDIBits` to grab screen pixels. The output is next passed through a JPEG encoding library to construct an image the client's browser can download and render. These encoded JPEG screenshots are delivered to the client through the internal Web server running on Flashproxy. Figure 5 illustrates this process.

The client initializes AJAX remote display by declaring an HTML image that points to our remote display server. This causes the browser to issue an HTTP request to download the first screenshot and render it. To force the image to refresh, the AJAX component set the image's `onload` property to execute an image update routine as soon as the image finishes rendering. The update function appends a special token to the image's URL, causing the browser to re-fetch the image. This approach makes refresh adaptive to both network transfer speeds and client rendering times; smaller displays will automatically refresh faster than larger displays, because they take less time to transfer and render. This approach works smoothly in practice and does not cause any display artifacts such as flickering images.

Some Flash programs, such as slideshows, change their graphical display infrequently. In such cases, the constant refreshing as described above would waste bandwidth transferring the same image many times. To remedy this, the remote display server delays its response to the client's image fetch request until it takes a screenshot that the client has not previously seen. Screenshot comparison is done by hashing images.

The second major component of remote display traps client's user input and forwards it to the remote display server. Fortunately, JavaScript provides facilities to interpose on the essential forms of user input, including mouse movement, mouse clicks, and keyboard input. Flashproxy's client-side code installs hooks that catch user input events and issue a special RPC call to the RPC nexus. The RPC nexus forwards these events to the remote display server, which synthesizes them in the doppelbrowser.

One important user input consideration is focus. Not all input on the Web page is destined for Flash programs; for example, typing in a HTML text field co-located on the same Web page should not expose those keystrokes to Flashproxy. Therefore, Flashproxy only processes input events that fire while a Flash program is in focus. A Flash program *receives* focus when the user moves the mouse over it, and *loses* focus when the mouse is moved elsewhere. Flashproxy conveniently utilizes JavaScript's `onmouseover` and `onmouseout` properties for the remote display image to keep track of which Flash program is in focus, if any. Each input event can then be filtered, and only input destined for Flash will be sent to the appropriate doppelbrowser.

4.4 JavaScript code injected into the user's browser

To implement some of its functionality, Flashproxy injects special JavaScript code into Web pages that flow through it. Some of this code contains the logic that periodically looks for new Flash content, as described in section 3.4. Flashproxy installs the code as a JavaScript `window.onload` event that fires when the page finishes rendering. The Web page itself might have used `window.onload` for its own purposes; therefore, we tailpatch by saving the previous function contained in `window.onload` and executing it before running our code. To ensure that the Web page cannot overwrite `window.onload` containing our logic, Flashproxy injects this code into the bottom of the page.

As discussed in section 3.4, Flashproxy must fool the client's browser into believing that it has runtime support for Flash.

Category	# of programs	# successfully run by Flashproxy
Navigation	16	16 (100%)
Interactive features	15	13 (87%)
Advertisements	11	10 (91%)
Games	13	13 (100%)
Static images and slideshows	23	23 (100%)
Multimedia streaming	12	12 (100%)
Animation	10	10 (100%)

Table 1: Summary of results. We manually classified 100 randomly selected Flash programs found by crawling the Web. Next, we disabled Flash on a Firefox browser, and tested whether these 100 programs (and the pages that embed them) were successfully accessible using Flashproxy.

The most widely-used method to check for plug-in support involves checking either the `navigator.plugins` or the `navigator.mimeTypes` arrays provided by the browser's JavaScript runtime. Unfortunately, we cannot insert new objects into these structures because the browser disallows such modifications. A complete solution would have to add an interposition layer to the Web page code in the spirit of BrowserShield [21], and intercept all references to these two arrays. However, as a practical and effective solution that works for most Web content, we instrumented the proxy to look for "`navigator.plugins`" and "`navigator.mimeTypes`" strings in Web pages, and rewrite them into strings that refer to shadow copies of these arrays. Shadow plug-in arrays have the same content as the originals plus a new element representing a fake Flash plug-in. Flashproxy initializes the shadow plug-in arrays at the top of Web pages, so that any detection code within the Web page can find these structures.

The detection code is implemented in 114 lines of JavaScript, translating into a small 3.3KB increase in Web page size. When a Flash object is found, the detection code will fetch and inject additional code that implements the client's RPC layer and AJAX remote display. Flashproxy's total client-side code when rendering Flash consists of 484 lines or 10KB, which is remarkably small considering the functionality it provides.

5. EVALUATION

In this section, we evaluate the effectiveness and performance of our Flashproxy architecture and prototype. We first test whether Flashproxy works in practice for Flash programs encountered in real Web content. Next, we measure the performance of the three major Flashproxy components: the binary rewriter, the RPC layer, and the AJAX remote display. Finally, we briefly demonstrate the security benefits of our system by showing that Flashproxy is able to isolate the user's browser from a Flash-based denial of service attack.

5.1 Does Flashproxy work?

To understand how well Flashproxy works on real Web content, we used the Heritrix crawler [15] to find 2,100 Flash programs on the Web. We randomly selected 100 programs and manually inspected them. To understand the different ways in which Flash is used, we classified these 100 programs into seven categories: Flash-based Web page navigation, interactive features such as a scoreboard or map, ads, games, static images or slideshows, multimedia streaming applications such as YouTube, and finally, non-interactive Flash animation.

Flash version	3	4	5	6	7	8	9
# programs	1 (0.05%)	40 (1.9%)	180 (8.6%)	670 (32%)	449 (21%)	707 (34%)	53 (2.5%)

Table 2: Flash versions. This table shows the number of Flash programs of each version found during our crawl.

Table 1 shows the result. We found Flash programs evenly spread through all categories, though images and slideshows were the most common. We also analyzed all crawled Flash programs to test which Flash version they were compiled with. Table 2 shows that most Flash programs are version 5, 6, 7, or 8; older versions are rare, as are newer Flash 9 programs.

Next, we tested whether Flashproxy was able to handle these 100 programs. To do this, we disabled Flash on a Firefox browser, and then used Flashproxy to browse through the Web pages hosting these programs. Table 1 shows the number and percentage of programs in each class that worked through Flashproxy. Flashproxy was able to identify, rewrite, remotely execute, and remotely display 97 of the 100 programs.

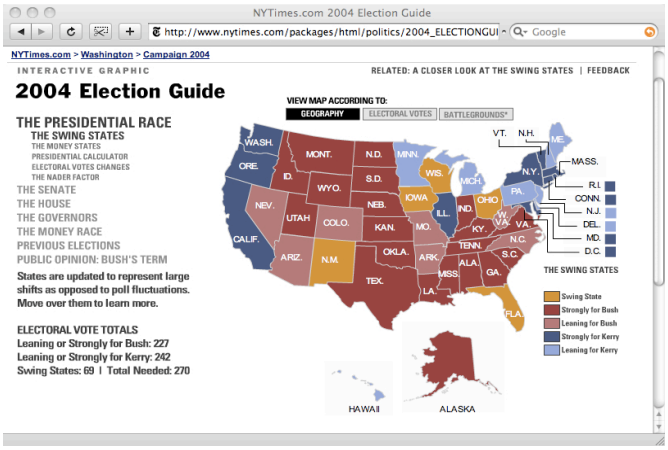
Flashproxy failed to render three pages correctly. The first was an SSL-encrypted HTTPS page, preventing the proxy from getting access to the Web page content. The second contained buggy JavaScript code. The Web page's code throws an exception on load, preventing our embedded detection and RPC code from executing. Strictly speaking, this page did run correctly by crashing, since the page itself is broken. The third page contained a Flash ad, but used a method for testing for Flash support that our injected JavaScript detection code missed. We have since added support for this method, but more generally, we believe we would have to use BrowserShield-like techniques [21] to catch and fool all JavaScript-based mechanisms for testing Flash support.

We also tested these 100 programs with a Mobile Safari browser running on an iPod Touch (iTouch). Overall, the iTouch was able to render and remote display the same programs as was Firefox. Figure 6 compares a New York Times Web page rendered on a desktop Safari browser with Flash support enabled, with an iTouch-based Safari browser that lacks Flash support visiting the same page through Flashproxy.[1] Flashproxy lets a user display and interact with the Flash content using their iTouch, and had the iTouch user visited the page without Flashproxy, their browser would have displayed an error message indicating a lack of Flash support.

Our evaluation did uncover some limitations. One Flash "greeting card" program asked the user to cut and paste a link to the card, but we have not attempted to integrate the clipboard of the doppelbrowser with that of the user's browser. Some Flash objects contained audio streams, and our remote display AJAX component does not currently support audio. Fixing both of these shortcomings is possible, but left as future work.

Our iTouch evaluation also uncovered some user interface issues specific to the iTouch's touchscreen interface. For example, the iTouch does not have a notion of mouse movement, because the corresponding gesture controls the panning of the page instead. As a result, some games which rely on dragging or moving the mouse were limited in their use. As well, keyboard input on an iTouch is troublesome, as Mobile Safari only displays the touch-based keyboard when a user clicks on a text field in a form.

[1] Note: this screenshot was taken using the iPhone simulator that ships with the iPhone SDK. The display of the browser on the simulator is indistinguishable from that of the iTouch device itself.



(a) browsing using desktop Safari with Flash enabled

(b) browsing using iTouch Safari through Flashproxy

Figure 6: Screenshots. (a) A desktop Safari browser with Flash enabled viewing a New York Times page with a mixture of HTML and Flash. (b) The iTouch Safari browser viewing the same content through Flashproxy.

5.2 Binary rewriting

Flashproxy uses binary rewriting to interpose on interactions between Flash and its execution environment. We evaluated Flashproxy to answer the following questions:

- How often are the interposed functions found in real Flash programs?
- How long does binary rewriting take in practice, and how often does rewriting fail?
- How much execution overhead is experienced by rewritten Flash programs?

To answer these questions, we analyzed the remaining 2,000 Flash programs gathered during the Web crawl.

5.2.1 *Frequency of interposed functions*

Table 3 shows the percentage of programs that contain each of the functions or classes that Flashproxy interposes upon while binary rewriting; Section 3 discusses these functions. For example, `ExternalInterface` methods were encountered in 11% of crawled Flash programs, while we didn't find any programs that used `FileReference`. Printing from Flash is similarly rare, whereas redirecting a browser to a URL is common, being present in 52% of Flash programs.

instruction, function, or class	# programs found that contain it
GetURL	1041 (52%)
LoadMovie	470 (24%)
LocalConnection	308 (15%)
ExternalInterface	228 (11%)
PrintJob	2 (0.1%)
FileReference	0 (0%)

Table 3: External interactions in Flash programs. This table shows the number of Flash programs found in our crawl that contain each each instruction, function, or class associated with an external interaction that Flashproxy must trap.

5.2.2 *Latency and success rate of rewriting*

Next, we tested how long it takes to rewrite flash programs. We found this was fast; on average, rewriting a program took 1.26 seconds on a 2.8GHz, dual-core Intel Xeon using Linux. Thus, binary rewriting will not significantly affect the startup latency of Flash content. Binary rewriting does inflate the size of Flash programs, as we add bytecode instructions to every function invocation within the program. In spite of this, the size of the rewritten Flash programs increased by only 3.7% on average: the media content within Flash programs dominates the bytecode. As well, although rewriting might increase code size more for function-call intensive programs, this would still not lead to any additional transfer time, since a rewritten Flash program is served to the doppelbrowser over a local connection rather than to the user's browser over the wide-area network.

We encountered 9 Flash programs (0.45%) that we could not rewrite. These programs used code obfuscation and decompilation prevention techniques that crashed the Flasm disassembler. We should be able to improve the reliability of the disassembler to handle this code, given that the Flash Player itself is able to disassemble and execute them, however we have not tried to do so.

5.2.3 *Runtime overhead introduced by rewriting*

Because we inject additional instructions into a rewritten program, it could experience runtime overhead and lower performance. To quantify this, we wrote a Flash program that calls a null function 100,000 times, and measured its execution time before and after binary rewriting. We found that each direct function call takes 11 microseconds, whereas a rewritten function call takes 26 microseconds. Although this seems like a significant slowdown, the small code size increase measured in Section 5.2.2 suggests that most instructions in Flash are not function calls, and that the frame rate and execution speed of Flash is usually dominated by graphics operations rather than frequent function calls.

5.3 RPC layer

Flashproxy's JavaScript RPC layer forwards external calls between Flash programs running in the doppelbrowser and the user's browser. We now quantify the overhead of this call forwarding path, broken down across the Flashproxy's components.

To do this, we constructed a microbenchmark Flash program that measures the latency of using `ExternalInterace` to invoke a JavaScript function `foo()` defined by the Web page in which the Flash program is embedded. This function accepts a single string

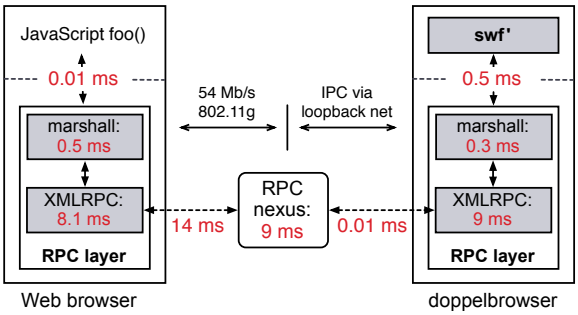


Figure 7: Flash-to-JavaScript RPC overhead. The total call time from (including return) from Flash to a remote JavaScript function is 41.4 ms, compared to 0.46 ms for a direct call from Flash to JavaScript on the same browser.

Flash program	category	size	frame rate (laptop)	frame rate (iTouch)
Interactive navigation pane	navigation	217x353	16 fps	7 fps
large horizontal ad banner	ads	728x90	15 fps	7 fps
small side ad banner	ads	350x40	21 fps	11 fps
YouTube video streaming	multimedia	497x418	14 fps	5 fps
Funnyplace.org video streaming	multimedia	320x260	20 fps	7 fps
cartoon (using vector animation)	animation	490x340	12 fps	5 fps
maze highly-interactive game	games	817x719	6 fps	2 fps
poker game	games	567x423	10 fps	4 fps
full-screen highly-animated page	feature	942x673	3 fps	2 fps

Table 4: Remote display performance. Flashproxy achieves good frame rates for most categories, especially ads and navigation. The performance of the system degrades for highly-interactive programs with a big display area.

argument and returns another string. We rendered the Web page first using Firefox on a 2.4GHz Intel Core 2 Duo running Mac OS X, and next using the same Firefox browser and client computer but with Flash disabled and Flashproxy enabled. Flashproxy was running on a 2.8GHz Intel Xeon server running Linux, and the client computer was connected to it and to the Web server over a 54 Mb/s 802.11g wireless LAN.

Our measurements show that a direct Flash-to-JavaScript call (including return) takes 0.46 ms natively, while the same call takes 41.4 ms with Flashproxy, for an overhead of 40.9 ms. Figure 7 breaks down this overhead across the Flashproxy components. The most expensive operations are the 14 ms network round-trip time from our client to the RPC nexus, and the delays incurred internally by the browser's XMLRPC engine. As well, the RPC nexus takes on average 4.5 ms to forward a call. We have not yet tried to optimize the RPC nexus, and as such, its network and data forwarding operations are quite inefficient. With some optimization we believe we could remove most of the overhead in the RPC nexus.

The extra instructions inserted by our binary rewriter did not significantly increase the latency of the `ExternalInterface` call from Flash to JavaScript. This call took 0.46 ms natively and 0.5 ms with binary rewriting. Finally, argument marshaling and unmarshaling within the JavaScript RPC layer incurred some overhead as well. We found that most of this overhead is attributable to the JSON library we use.

Overall, we feel that 41.4 ms is acceptable for the RPC layer. Flash-to-JavaScript calls are typically used to redirect the user's browser to a new Web page, or to pass to Flash text typed by the user. For such infrequent uses, the overhead will be unnoticeable to the user. However, Flashproxy could slow down Web applications that constantly exchange data between Flash and JavaScript, although we believe this usage scenario is uncommon.

The RPC layer on the client browser is also responsible for forwarding user input, such as mouse movements or key-presses. Forwarding requires an XMLRPC call to the RPC nexus, which then executes the input event in the doppelbrowser. We found that it takes 4 ms to transfer the event through the client browser's XMLRPC layer, 7 ms to transfer it to the RPC nexus over a wireless network, and 1 ms to synthesize the event in the doppelbrowser, for a total user input latency of 12 ms.

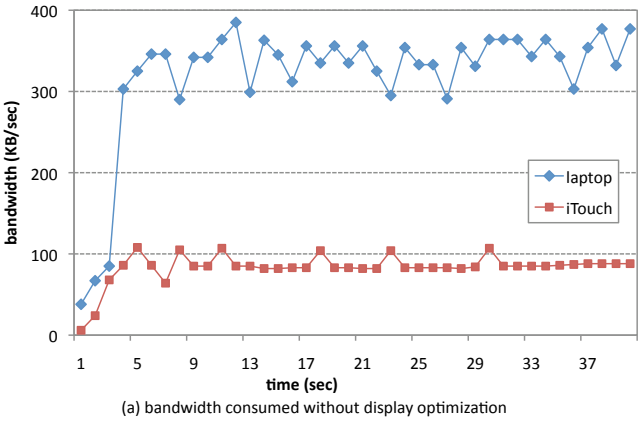
5.4 Remote display

The AJAX remote display component used by Flashproxy is based on transferring JPEG-encoded screenshots from the doppelbrowser, as described in Section 4. This simple approach works surprisingly well for many types of Flash content. To quantify the performance of our remote display implementation, we chose several representative Flash programs from the categories defined in table 1 and measured the performance the client experienced while browsing through Flashproxy. We tested two client configurations: a laptop with a Firefox Web browser with Flash disabled on a 2.4GHz Intel Core 2 Duo running Mac OS, and an iTouch with a Mobile Safari browser (which lacks Flash support). Both clients connected to the Internet over a 54 Mb/s 802.11g network.

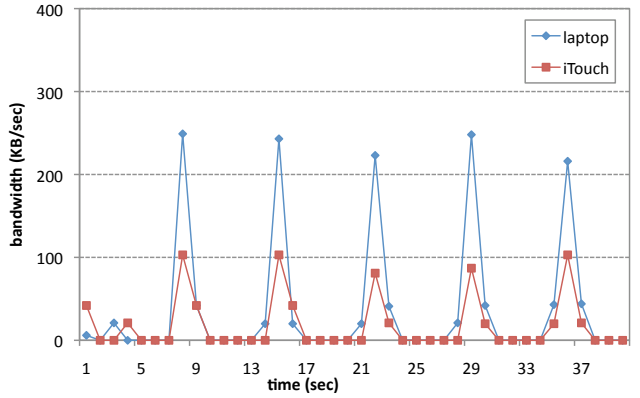
Table 4 summarizes our results. For each Flash program, we show the effective frame rate at which the client was able to view it. We see that Flashproxy does well for most of the categories we have examined. Small-area Flash programs such as ads render very quickly on the desktop; the user experiences no noticeable display lag. Larger animations and streaming video render fast enough for the user to comfortably see the content, though there is a noticeable reduction in frame rate. Full-screen, highly-animated Flash pages performed poorly; their screenshots were large and changed on every frame, taxing the client's rendering engine and incurring large transfer times. Highly-dynamic games experienced the same problem, though games with a lower screenshot refresh rate such as poker were quite playable. We did not evaluate static images and slideshows, since these Flash programs only need one or a few very infrequent display updates, and Flashproxy can trivially handle them.

Although the iTouch performed 2x-3x slower on average than the desktop, many Flash programs still performed surprisingly well on the mobile device. For example, navigation panes, ads, and smaller-size movies all achieved good responsiveness. In addition, our remote display optimization of eliminating redundant screen updates frequently masks the slower performance of the iTouch, because with no outstanding screen updates, the client does not consume any bandwidth or rendering resources.

Figure 8 illustrates this by showing the bandwidth consumed by a slideshow Flash program on an iTouch and our laptop, with and without our display optimization. For each elapsed second, the y-axis shows bandwidth consumed in that second. Note that the clients are the bottleneck; Flashproxy can always generate display screenshots faster than the clients can consume them. In 35 of the 40 seconds shown, the number of screen updates is small, and the display performance is identical on the laptop and the iTouch. The spikes at the remaining 5 seconds represent slide transition effects. Rapid display updates tax the client's CPU, and as the result the less powerful iTouch renders fewer display updates than the laptop. The



(a) bandwidth consumed without display optimization



(b) bandwidth consumed with display optimization

Figure 8: Bandwidth consumed while rendering a slideshow Flash program. With no display optimization, the laptop and the iTouch fetch screen updates as fast as they can render them; the laptop consumes 2-3x more bandwidth than an iTouch. However, most screen updates are unnecessary as they duplicate previous updates; our display optimization eliminates such updates and drastically reduces consumed bandwidth.

available bandwidth is far from being saturated by either the laptop or the iTouch. The graph also shows that our display optimization saves a significant amount of bandwidth: the iTouch consumes 18 KB/s average bandwidth to render the slideshow, while the laptop uses 37 KB/s, both lower than 83 KB/s and 320 KB/s they use without the optimization.

While testing remote display, we noticed that the JPEG compression settings used to take screenshots can be tuned to improve frame rates. Aggressive JPEG compression can dramatically reduce file sizes and thus network transfer time, at the cost of display quality. For example, Figure 9 shows the frame rate for a YouTube video displayed through Flashproxy as we vary the JPEG quality knob. With no compression, the frame rate is 4.7 fps, but with compression set at 60%, it is nearly three times higher. Depending on the type of Flash program, we could tune JPEG compression to provide the best balance of quality and performance. For example, higher frame rate is more important in streaming video, while slideshows or mostly static navigational Flash require higher-quality display. Exploring dynamic quality adaptation is future work. For our experiments, we set the JPEG quality to 60%, which achieves a good balance between quality and performance.

Remote display consumes a different amount of network bandwidth than a native Flash program. We measured the amount of traffic consumed by a bandwidth-intensive Flash program – a 38-second YouTube video. To render this video with a native Flash plug-in, a client will transfer the full 1.5 MB video. Spread over 38 seconds, this averages to 40 KB/s, though in practice a client will transfer the video as quickly as possible. With Flashproxy, a client receives display updates at 14 frames per second, resulting in 81 KB/s of steady-state bandwidth and a total transfer of 3.1 MB over the lifetime of the movie. Given the unoptimized nature of our remote display mechanism, this overhead seems acceptable. We have not yet explored alternate remote display implementations that would allow us to decrease bandwidth consumption, though related work suggests this is possible [17].

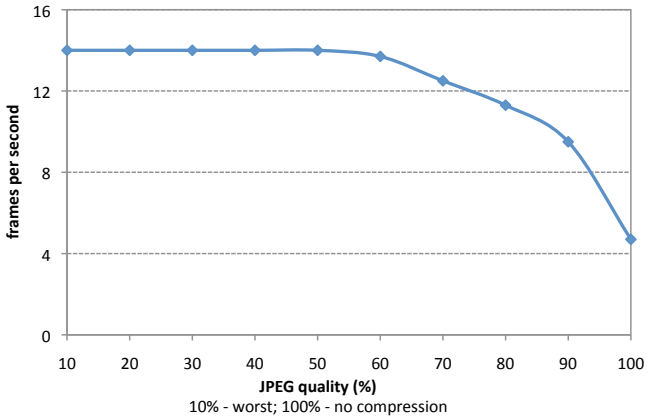


Figure 9: YouTube video frame rate vs. JPEG quality. More compression produces smaller screenshots, improving frame rate but decreasing quality. Our display optimization was enabled during this experiment, but ineffective as every frame of the video was distinct.

5.5 Security

One benefit of Flashproxy is that the Flash Player is isolated from the client in a remote sandbox. Thus, any vulnerabilities in the Flash player cannot be used to affect the client. Although we could not find a malicious Flash program to test, there have been public reports of Flash vulnerabilities and the existence of exploit programs [26]. We did, however, find a Flash program that exploited a bug in the Flash plug-in to perform a denial of service attack and either crash or hang the user's browser [25]. We verified that by using Flashproxy, the client avoids these attacks. Instead, the attacks affect the doppelbrowser; however, Flashproxy can easily destroy the virtual machine sandbox enclosing the doppelbrowser to deal with such an attack.

5.6 Summary

In this section, we evaluated the functionality and performance of Flashproxy using a combination of real-world Flash programs and microbenchmarks. Flashproxy correctly rendered nearly all content we exposed it to. The AJAX remote display performs well for many classes of Flash programs, such as dynamic navigation menus, slideshows, ads, and even YouTube streaming videos, where it achieved a display throughput of 14 frames per second. We also demonstrated the security benefits of Flashproxy, showing that a malicious Flash program was isolated from the client.

6. EXTENDING FLASHPROXY BEYOND FLASH

Flashproxy currently supports only Flash programs. As a thought exercise, we considered whether Flashproxy could be extended to support other forms of active Web content. In particular, we examined what it would take to support Microsoft's Silverlight, an emerging standard for rich multimedia Web applications.

Several components of Flashproxy would stay virtually unchanged, including the remote display, user input forwarding, Web proxy, DOM-based plug-in object detection, and client AJAX code. Supporting Silverlight would require us to implement a new binary rewriting engine for Silverlight code, and to figure out how to use binary rewriting to interpose on interactions between Silverlight and the browser.

Silverlight is currently available in two versions. In version 1.0, a Silverlight program consists of an object description in a text-based XAML markup language, which defines the graphics, audio, and video components of the application, and JavaScript code, which is defined in the Web page hosting the application. Similar to HTML, XAML content does not compute anything by itself. Instead, it uses JavaScript on the hosting page to handle certain events, such as user input.

To support Silverlight 1.0, Flashproxy would need a new text-based rewriting engine, but it could reuse all of its other components. The rewriting engine would find all references to JavaScript functions in XAML files and rewrite them into stubs that will use Flashproxy's RPC layer to forward calls from the doppelbrowser to the client browser. Unlike Flash, XAML files are not binary and should be easily parsable. In addition, JavaScript on a hosting page can access elements defined in a XAML file, for example by finding the application's plugin instance in the DOM and using "content.FindName". Such references could be rewritten similarly to `ExternalInterface.addCallback`'s implementation for Flash.

Silverlight 2.0 (currently in beta) adds support for application code written in Microsoft's .NET Framework languages, such as C#. Like Flash programs, Silverlight 2.0 .NET applications run in a sandbox that provides the necessary .NET runtime support and enforces security policies. Silverlight 2.0 ships with a lightweight class library that is considerably smaller than .NET Framework's base class library. It prohibits use of security-sensitive .NET components, such as local file I/O, and replaces them with safe equivalents, such as the `System.IO.IsolatedStorage` classes. .NET-based Silverlight 2.0 applications contain a binary component that specifies the code to be executed in MSIL, a low-level language similar to Java bytecode.

Like Flash, Silverlight 2.0's runtime supports classes and methods that access or modify external state and therefore must be rewritten by Flashproxy. Thus, Flashproxy would need another binary rewriter for MSIL. Fortunately, we can leverage many existing tools, including Microsoft's own MSIL assembler and disassembler.

Many external-state classes of .NET are already disallowed by Silverlight's security policy. We anticipate the vast majority of the remaining changes would be located in the `System.Windows.Browser.*` classes, which allow Silverlight applications to programmatically access and manipulate the Web page's DOM. For example, the HtmlDocument or HtmlElement Silverlight classes have methods corresponding to JavaScript's native methods for document and element objects; e.g., HtmlDocument's `GetElementByID()` corresponds to JavaScript's `document.getElementById()`. When an application invokes such a method, Silverlight marshals the arguments and executes the corresponding native JavaScript function in the browser. Flashproxy would need to rewrite all of these DOM-related methods into its own stubs, which would then relay the execution from the doppelbrowser into the client browser.

The existing RPC layer implementation could be mostly reused. One change we would need to make is to add support for Silverlight's multithreading. Flash programs are single-threaded, and therefore our RPC layer currently only supports one outstanding call. It should be straightforward to extend the RPC layer with buffering and queueing to support multiple outstanding calls.

7. RELATED WORK

Flashproxy builds upon ideas from three research areas: the use of proxies to support mobile devices and software, applications of binary rewriting, and the use of remote display. We discuss each of these areas in turn.

7.1 Proxies for supporting mobile devices

Many research projects and commercial systems use proxies to support mobile computing. Proxies are generally used to adapt content to better suit the network, hardware, and software characteristics of the remote device. Proxies have been used to adapt Web content for mobile devices [2 9 10], to bridge heterogeneous multicast groups together and adapt data flowing between them [4], to enable collaboration and document editing on bandwidth-limited devices [6], and to facilitate application-specific control and data adaptation by using existing application component APIs [7].

Brooks et al. define the notion of an HTTP stream transducer [3]; a transducer is a generalization notion of a Web proxy that views and potentially alters HTTP content as it flows between servers and clients. Zenel discusses a proxy-based filtering architecture that supports the dynamic, transparent insertion of application and protocol specific filters [34].

Flashproxy can be seen as an example of a stream transducer or a point instantiation of Zenel's filtering architecture. More broadly, Flashproxy is similar to all of this earlier work on proxy-based content adaptation for mobility, but it focuses on the mechanisms needed to support active content, such as Flash.

7.2 Binary rewriting

Binary rewriting has been used in many application domains, including software-based fault isolation [33], virtualization [1], and code obfuscation to prevent reverse engineering [19]. Toolkits exist to facilitate binary rewriting on particular hardware architectures; for example, Diablo provides a retargetable link-time binary rewriting framework, supporting ARM, i386, IA64, Alpha, and MIPS architectures [32]. Diablo has been used to enable applications such as program compaction, performance optimization, obfuscation, and instrumentation. The goal of binary translation, a variant of binary rewriting, is to convert programs compiled for older systems to new architectures [28].

Binary rewriting techniques have been applied to bytecode-oriented programs [5 18]. Naccio uses binary rewriting to enforce safety policies expressed in a high-level language on Java bytecode [8]. Similar in some regards to Flashproxy, Rico [29] and PBJars [30 31] use a Web proxy to mediate all HTTP requests from a population of Web browsers, identifying embedded Java programs. Like Naccio, these systems use binary rewriting to enforce additional security policies on the Java programs to improve end-host security.

DVM uses a proxy to intercept Java code downloads in an enterprise or other organizational setting, and uses binary rewriting to factor out Java code verification, security services, and remote monitoring [27]. Like Flashproxy, DVM eliminates complexity

from end hosts; unlike Flashproxy, DVM still requires a bytecode virtual machine on end hosts to execute the rewritten Java code.

Simple Flash binary rewriting tools exist, but they are mostly used as code obfuscators to prevent the reverse engineering of ActionScript programs [13]. In contrast, Flashproxy relies on Flash binary rewriting to interpose on the external interactions of a Flash program. Flashproxy builds on the open-source Flasm assembler and disassembler [14] to implement its binary rewriter.

7.3 Remote display

Remote display has been used in many systems to provide location transparency or remote access. The X windows system was one of the earliest to support remote display, and it permits individual applications' interfaces to be projected across a network [23]. VNC is a virtual screen buffer, permitting the entire windowing system of an operating system to be remotely displayed [22]. Many commercial systems have improved the performance and reliability of remote display, including Sun Ray [24], Citrix [20], and Microsoft's RDP.

Remote display has been adapted to mobile and low bandwidth environments. For example, low-bandwidth X (LBX) uses a caching and compression proxy server deployed at X clients to reduce the amount of traffic sent to X servers [11]. More recently, pTHINc [17] explored remote display mechanisms for enabling Web browsing on mobile, wireless devices such as PDAs. pTHINc demonstrates that it is possible to provide rich, usable remote display on constrained devices for the kinds of content types and applications that Flashproxy is likely to encounter.

8. CONCLUSIONS

In this paper, we described a proxy-based technique for transparently supporting active content, such as Flash or Silverlight, on mobile browsers that do not have built-in support for that content. Our approach relies on the proxy to splice out the embedded active content from the Web page and to replace it with an AJAX-based remote display Web component. The spliced out active content is executed instead in a sandbox on the remote proxy. We designed, implemented, and evaluated Flashproxy, a prototype of our approach that provides Flash support to any browser that has JavaScript and sufficient bandwidth.

A challenge to getting our approach to work is handling interactions between a Flash program and its execution environment, including calls made between a Flash program and JavaScript functions on the embedding page, or directives issued from Flash to the browser to navigate to new pages. To handle these interactions, Flashproxy uses binary rewriting to insert interposition hooks on the relevant Flash function and method calls. Flashproxy relays these calls to the user's browser with a JavaScript-based RPC layer.

We evaluated Flashproxy using a combination of microbenchmarks and manual examination of Web pages containing Flash programs discovered using a Web crawler. Our evaluation shows that Flashproxy is able to execute virtually all Flash programs we encountered successfully, and that the resulting system has adequate performance for many classes of Flash content. As well, we demonstrated that Flashproxy is able to isolate the user and her browser from security flaws in Flash programs or within the Flash runtime environment itself. Overall, our experience with Flashproxy demonstrates the transparency and effectiveness of our approach.

9. ACKNOWLEDGMENTS

We thank Charles Reis for his help in understanding the intricacies of browsers, JavaScript, and the DOM. We also thank our shepherd, Matt Welsh, and our anonymous reviewers, whose feedback helped us to improve the quality of this paper. This research was supported in part by the National Science Foundation under grants CNS-0132817, CNS-0430477, and CNS-0627367, by the Torode Family Endowed Career Development Professorship, and by the Wissner-Slivka Chair.

10. REFERENCES

[1] ADAMS, K., AND AGESEN, O. A comparison of software and hardware techniques for x86 virtualization. In *Proceedings of the Twelfth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XII)* (Seattle, WA, March 2006).

[2] BICKMORE, T. W., AND SCHILIT, B. N. Digestor: Device-independent access to the world wide web. In *Proceedings of Sixth International World Wide Web Conference (WWW 1997)* (Santa Clara, CA, April 1997).

[3] BROOKS, C., MAZER, M. S., MEEKS, S., AND MILLER, J. Application-specific proxy servers as HTTP stream transducers. In *Proceedings of the Fourth International World Wide Web Conference (WWW 1995)* (Boston, MA, December 1995).

[4] CHAWATHE, Y., MCCANNE, S., AND BREWER, E. RMX: Reliable multicast in heterogeneous environments. In *Proceedings of IEEE INFOCOM 2000* (Tel-Aviv, Israel, March 2000).

[5] COHEN, G. A., CHASE, J. S., AND KAMINSKY, D. L. Automatic program transformation with JOIE. In *Proceedings of the 1998 USENIX Annual Technical Symposium* (New Orleans, LA, June 1998).

[6] DE LARA, E., KUMAR, R., WALLACH, D. S., AND ZWAENEPOEL, W. Collaboration and multimedia authoring on mobile devices. In *Proceedings of the First International Conference on Mobile Systems, Applications, and Services (MobiSys 2003)* (San Francisco, CA, May 2003).

[7] DE LARA, E., WALLACH, D. S., AND ZWAENEPOEL, W. Puppeteer: Component-based adaptation for mobile computing. In *Proceedings of the 3rd USENIX Symposium on Internet Technologies and Systems (USITS '01)* (San Francisco, CA, March 2001).

[8] EVANS, D., AND TWYMAN, A. Flexible policy-directed code safety. In *Proceedings of the 1999 IEEE Symposium on Security and Privacy* (Oakland, CA, May 1999).

[9] FOX, A., AND BREWER, E. A. Reducing WWW latency and bandwidth requirements by real-time distillation. In *Proceedings of the Fifth International World Wide Web Conference (WWW 1996)* (Paris, France, May 1996).

[10] FOX, A., GOLDBERG, I., GRIBBLE, S. D., AND LEE, D. C. Experience with Top Gun Wingman: A proxy-based graphical web browser for the 3Com PalmPilot. In *Proceedings of the Middleware '98* (Lake District, England, September 1998).

[11] FULTON, J., AND KANTARJIEV, C. K. An update on low bandwidth X (LBX). *The X Resource*, 5 (1993), 251–266.

[12] GARRETT, J. J. Ajax: A new approach to Web applications. `http://www.adaptivepath.com/publications/essays/archives/000385.php`, February 2005.

[13] HAYDEN, D. OBFU: A Flash Actionscript obfuscator. `http://www.opaque.net/~dave/obfu/`, 2001.

[14] HAYDEN, D., MORTON, D., KOGAN, I., AND ZHEN, W. The Flasm command line assembler/disassembler of flash actionscript bytecode. `http://www.nowrap.de/flasm`, 2007.

[15] INTERNET ARCHIVE. The Heritrix Web crawler project. `http://crawler.archive.org/`.

[16] JSON. `http://www.json.org/`.

[17] KIM, J., BARATTO, R. A., AND NIEH, J. pTHINC: A thin client architecture for mobile wireless web. In *Proceedings of the Fifteenth International World Wide Web Conference (WWW 2006)* (Edinburgh, Scotland, May 2006).

[18] LEE, H. B., AND ZORN, B. G. BIT: A tool for instrumenting Java bytecodes. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems (USITS '97)* (Monterey, CA, December 1997).

[19] LINN, C., AND DEBRAY, S. Obfuscation of executable code to improve resistance to static disassembly. In *Proceedings of the 10th ACM Conference on Computer and Communications Security (CCS 2003)* (Washington, DC, October 2003).

[20] MATHERS, T. W., AND GENOWAY, S. P. *Windows NT Thin Client Solutions: Implementing Terminal Server and Citrix MetaFrame*. Macmillan Technical Publishing, Indianapolis, IN, November 1998.

[21] REIS, C., DUNAGAN, J., WANG, H. J., DUBROVSKY, O., AND ESMEIR, S. BrowserShield: Vulnerability-driven filtering of dynamic HTML. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI '06)* (Seattle, WA, November 2006).

[22] RICHARDSON, T., STAFFORD-FRASER, Q., WOOD, K. R., AND HOPPER, A. Virtual network computing. *IEEE Internet Computing 2*, 1 (1998), 33–38.

[23] SCHEIFLER, R. W., AND GETTYS, J. The X window system. *ACM Transactions on Graphics (TOG) 5*, 2 (April 1986), 79–109.

[24] SCHMIDT, B. K., LAM, M. S., AND NORTHCUTT, J. D. The interactive performance of SLIM: a stateless, thin-client architecture. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP '99)* (Kiawah Island Resort, SC, December 1999).

[25] SECUNIA. Macromedia Flash ActiveX denial of service. `http://secunia.com/advisories/7545/`, November 2002.

[26] SECUNIA. Adobe Flash Player multiple vulnerabilities. `http://secunia.com/advisories/26027/`, July 2007.

[27] SIRER, E. G., GRIMM, R., GREGORY, A. J., AND BERSHAD, B. N. Design and implementation of a distributed virtual machine for networked computers. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP '99)* (Kiawah Island Resort, SC, December 1999).

[28] SITES, R. L., CHERNOFF, A., KIRK, M. B., MARKS, M. P., AND ROBINSON, S. G. Binary translation. *Communications of the ACM 36*, 2 (February 1993).

[29] SONG, Y., AND FLEISCH, B. D. Rico: A security proxy for mobile code. *Journal of Computers and Security 23*, 4 (2004), 338–351.

[30] SONG, Y., AND FLEISCH, B. D. Utilizing binary rewriting for improving end-host security. *IEEE Transactions on Parallel and Distributed Systems 18*, 12 (December 2007), 1687–1699.

[31] SONG, Y., XU, Y., AND FLEISCH, B. D. Design and performance evaluation of a proxy-based Java rewriting security system. In *Proceedings of the 26th IEEE International Conference on Distributed Computing Systems, (ICDCS 26)* (Lisboa, Portugal, July 2006).

[32] VAN PUT, L., CHANET, D., DE BUS, B., DE SUTTER, B., AND DE BOSSCHERE, K. DIABLO: a reliable, retargetable and extensible link-time rewriting framework. In *Proceedings of the 2005 IEEE International Symposium On Signal Processing And Information Technology* (Athens, Greece, December 2005).

[33] WAHBE, R., LUCCO, S., ANDERSON, T. E., AND GRAHAM, S. L. Efficient software-based fault isolation. *ACM SIGOPS Operating Systems Review 27*, 5 (December 1993), 203–216.

[34] ZENEL, B., AND DUCHAMP, D. A general purpose proxy filtering mechanism applied to the mobile environment. In *Proceedings of the 3rd Annual ACM/IEEE International Conference on Mobile Computing and Networking (Mobicom '97)* (Budapest, Hungary, September 1997).