Pivot: Fast, Synchronous Mashup Isolation Using Generator Chains

James Mickens
Microsoft Research
mickens@microsoft.com

***Abstract*—Pivot is a new JavaScript isolation framework for web applications. Pivot uses iframes as its low-level isolation containers, but it uses code rewriting to implement synchronous cross-domain interfaces atop the asynchronous cross-frame `postMessage()` primitive. Pivot layers a distributed scheduling abstraction across the frames, essentially treating each frame as a thread which can invoke RPCs that are serviced by external threads. By rewriting JavaScript call sites, Pivot can detect RPC invocations; Pivot exchanges RPC requests and responses via `postMessage()`, and it pauses and restarts frames using a novel rewriting technique that translates each frame's JavaScript code into a restartable generator function. By leveraging both iframes and rewriting, Pivot does not need to rewrite all code, providing an order-of-magnitude performance improvement over rewriting-only solutions. Compared to iframe-only approaches, Pivot provides synchronous RPC semantics, which developers typically prefer over asynchronous RPCs. Pivot also allows developers to use the full, unrestricted JavaScript language, including powerful statements like `eval()`.**

I. Introduction

A modern web application often contains JavaScript code from multiple origins. For example, a single web page might contain JavaScript from advertisers, analytics providers, and social networking sites. These external origins have differing levels of pairwise trust, and all of the external code must coexist with the JavaScript that belongs to the owner of the enclosing page. For security, each domain should expose a narrow interface to its code and data. Unfortunately, JavaScript has poor built-in mechanisms for encapsulation and information hiding (§II). Thus, to enforce strong isolation between untrusting origins, developers must use a *mashup isolation framework* [3], [14], [22], [24] to restrict cross-domain interactions.

A. Prior Isolation Schemes

At a high-level, these isolation systems use one of two approaches. Some use iframes as the fundamental isolation container. Browsers give each iframe a separate JavaScript runtime; each runtime has distinct global variables, heap objects, visual display areas, and so on. Iframes from different origins cannot directly manipulate each other's state—instead, they must communicate using the asynchronous, pass-by-value `postMessage()` call. By placing each domain's code in a separate frame, systems like Privilege-separated JavaScript [3] leverage fast C++ code inside the browser to enforce isolation boundaries. However, domains are forced to communicate via asynchronous message passing. Asynchronous channels are an unnatural fit for many types of cross-domain communication [22], and continuation-passing style (CPS) [18], the most popular method for converting asynchronous calls into pseudo-synchronous ones, can introduce subtle race conditions [18], [22].

Other mashup frameworks use a rewriting approach [22], [24]. In these systems, the integrating web page translates each domain's code into a constrained JavaScript dialect that lacks dangerous features like the `eval()` function. The rewriter also adds dynamic checks which enforce statically unverifiable security properties. Code from different origins runs in the same frame, so domains can communicate using synchronous interfaces that resemble traditional RPCs. From a developer's perspective, these synchronous interfaces are extremely attractive, since JavaScript's pervasive asynchrony is widely perceived to hinder application development and maintenance [25], [26], [27], [36]. Unfortunately, JavaScript is a highly dynamic language, and enforcing a reasonable isolation model requires the rewriter to insert runtime checks at every function call and property access. These checks, which are implemented in slow JavaScript instead of fast C++ inside the browser, can cause execution slowdowns of up to 10x compared to the original, untranslated source code [12], [22], [31].

B. Overview of Pivot

In this paper, we introduce Pivot, a new isolation framework that combines the performance of iframe solutions with the synchronous cross-domain interfaces that have traditionally been restricted to rewriting frameworks. Pivot uses iframes as isolation containers, and uses `postMessage()` as a low-level communication primitive. However, by combining a novel rewriting technique with dynamic patching of the JavaScript runtime [21], [23], Pivot provides true synchronous interfaces that avoid the potential race conditions of CPS's pseudo-synchrony.

Figure 1 depicts the architecture of a Pivot application. Pivot's trusted master frame places each untrusted domain into a separate "satellite" frame. Each satellite frame contains untrusted JavaScript code from an external domain, and an untrusted copy of the Pivot RPC library. Using that library, a satellite can register one or more public RPC interfaces with the Pivot master frame. Pivot implements a distributed directory service that allows domains to discover each other's entry points.

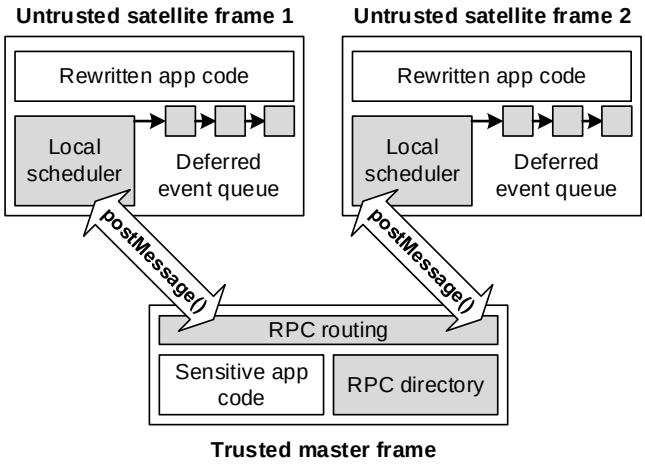

Figure 1: A sample Pivot application. Grey boxes indicate components that are implemented by Pivot.

```
//The yielding function.
function factorial(){
    var curr = 1;
    var n = 1;
    while(true){
        curr = curr * n;
        yield curr;
        n += 1;
    }
}

//Print the first 10 items in the factorial sequence.
var generator = factorial();
for(var i = 0; i < 10; i++){
    alert(generator.next());
}
```

Figure 2: Example of a JavaScript generator function.

To build a synchronous RPC interface atop the asynchronous `postMessage()` primitive, Pivot uses three techniques:

Static rewriting: Pivot rewrites each domain's JavaScript code, combining all of the code in a frame into a single *generator function* [9]. In contrast to a normal function, which uses the `return` statement to return a single value per function execution, and which loses its activation record upon returning, a generator uses the `yield` statement to remember its state across invocations. A generator can `yield` different values after each invocation. Figure 2 provides a simple example of a generator function that returns the factorial sequence. A rewritten Pivot frame is a generator function that starts execution when invoked by the local Pivot library, and yields to that library upon invoking an RPC.

Distributed scheduling: The untrusted Pivot libraries in each satellite communicate with the trusted Pivot library in the master frame. Collectively, the master library and the satellite libraries implement a distributed scheduler. When frame `X` invokes an RPC defined by frame `Y`, `X` yields control to its local Pivot library. That library sends a `postMessage()` to the master that contains the RPC name and the RPC arguments. The master uses `postMessage()` to forward the information to the Pivot library in `Y`. `Y`'s Pivot library invokes the appropriate code in Y, and uses `postMessage()` to return the result to the master. The master sends the result to the Pivot library in `X`. Finally, that library restarts `X`'s code by calling `X`'s generator function (Pivot places the result of the RPC in a well-known global variable so that `X`'s restarted generator can consume it). Buggy or malicious satellites may hang forever; to prevent denial-of-service attacks on RPC initiators, Pivot will timeout an RPC that takes too long to complete, restarting the initiator and forcing the RPC to return an error value.

Deferring asynchronous events: JavaScript is an event-driven language—programs define handler functions to deal with asynchronous inputs like mouse clicks and the arrival of network data. Browsers ensure that each frame is single-threaded and non-preemptable, i.e., only one event handler can run at any given time, and once started, a handler runs to completion in an atomic fashion. However, in Pivot, a frame's generator function can invoke an RPC (and thus yield) at arbitrary moments; while Pivot is waiting for the RPC response, the yielded frame may generate other asynchronous events. If Pivot allowed the associated event handlers to run, it would violate the guarantee of handler atomicity. Thus, Pivot dynamically interposes on the browser's event registration framework [23], wrapping each event handler in code that simply adds the real event handler to Pivot's deferred execution queue. Pivot detects when a satellite's generator yields at a "natural" termination point, i.e., a place at which the browser would normally declare a handler call chain to be finished. At these points, Pivot drains the deferred execution queue, executing any handlers that it finds.

At a high-level, Pivot resembles a user-mode threading library [32] in which frames are threads, and cross-frame RPCs cause threads to yield to the Pivot scheduler. However, Pivot must solve various challenges that are unique to the web environment, such as preserving the atomicity of rewritten event handlers without explicit support from the browser's JavaScript engine.

C. Our Contributions

Conventional wisdom is that mashup frameworks must use one of two approaches: fast but asynchronous frame-based isolation, or synchronous but slow rewriting-based isolation. Pivot is the first mashup framework that provides truly synchronous cross-domain interfaces while leveraging iframes as isolation containers. Pivot uses *less rewriting*

```
var X = {data: "hello"}; //Create a prototype object
                         //for a class named X.
var obj = {__proto__: X};//Create an instance of X.

alert(obj.data);         //Displays "hello".
X.data = "goodbye";
alert(obj.data);         //Displays "goodbye".
```

Figure 3: Example of JavaScript's prototype-based objects.

to provide *higher performance*—whereas prior rewriting schemes like Caja [24] and Jigsaw [22] require dynamic checks at every function call site and every `[]`-mediated property access, Pivot only requires checks at function call sites. Furthermore, since Pivot uses iframes as isolation containers, it allows the safe composition of rewritten code with unrewritten code. Domains that do not require the ability to make synchronous RPCs do not need to be rewritten. By including the (untrusted) Pivot satellite library, unrewritten satellites can still define externally visible RPCs; the unrewritten satellites will asynchronously serve those requests, and Pivot will pause and resume the rewritten callers as necessary, providing those callers with the desired synchronous RPC semantics. This safe composition of rewritten code and unrewritten code is much different than the rewrite-everything requirement of systems like Jigsaw, and it facilitates important performance optimizations when Pivot is run on browsers which have slow implementations of the `yield` statement (§V-C). Our empirical analysis of JavaScript call graphs demonstrates that Pivot's master/satellite architecture is well-aligned with the RPC patterns of real applications (§V-A).

Since each satellite frame contains an isolated JavaScript runtime, Pivot allows untrusted satellite code to use the full JavaScript language, including powerful functions like `eval()`. In contrast, rewriting systems that place all code in a single frame require applications to use a restricted JavaScript subset. Unlike prior rewriting systems that provide pseudo-synchrony [25], [26], [27], [36], Pivot provides *true* synchrony without adding special syntax for RPCs, or breaking the traditional JavaScript concurrency model of atomic handler execution. This makes it easier to write new Pivot applications and port legacy applications to Pivot. Our evaluation also shows that Pivot can provide an order of magnitude performance improvement over Jigsaw.

II. Background

In this section, we provide an overview of the JavaScript language, and describe how the JavaScript runtime interacts with the rest of the browser. We focus on the security interactions which influence Pivot's design (§III) and the design of previous isolation frameworks (§VI).

A. The JavaScript Language

JavaScript is an object-oriented scripting language that provides extensive runtime mechanisms for object reflection and mutation. JavaScript has little support for encapsulation or data hiding; this dearth of native isolation mechanisms, combined with an abundance of reflection interfaces, makes it difficult to write secure web applications that integrate code from mutually untrusting origins. For example:

```
//The "with" statement places a JavaScript
//object at the front of the name resolution
//chain.
var str = "global";
var obj = {str: "insideWithStmt"};
alert(str);    //Displays "global".
with(obj){
   alert(str); //Displays "insideWithStmt".
}

//The "delete" statement removes a property
//from an object. So, if a rewriter tries
//to hide sensitive global objects using
//with statements, attacker code can
//use delete statements to reveal the
//sensitive global.

//Original attacker code wants to send
//a network request.
delete XMLHttpRequest;
var ajax = new XMLHttpRequest(...);

//The rewritten code places the original
//attacker code inside a with statement,
//but the delete allows the attacker to
//access the real XMLHttpRequest!
var blinder = {XMLHttpRequest: null};
with(blinder){
    delete XMLHttpRequest;
    var ajax = new XMLHttpRequest(...);
}
```

Figure 4: An attacker can use statements like `delete` to subvert rewriting-based isolation.

Prototype poisoning: Traditional object-oriented languages like C++ use statically defined classes to implement object inheritance. In contrast, JavaScript uses dynamic prototype objects. As shown in Figure 3, a prototype is a JavaScript object whose property names define the properties belonging to all instances of that object. The property values of the prototype become the default property values for instances of that object.

An object's prototype is a mutable property called `__proto__`. By changing an object's `__proto__` value, one can change an object's type at runtime. In a prototype poisoning attack [1], [18], malicious code mutates the prototype for sensitive objects, or redefines their `__proto__` references to point to attacker-controlled objects. This allows the attacker to redefine the behavior of these sensitive objects. Poisoning attacks can be launched against application-defined objects, as well as the built-in objects provided by the browser. For example, by tampering with the prototype for the built-in regular expression object, an attacker can subvert security computations that look for patterns in strings.

Mutability attacks: Prototype poisoning is a specific example of the mutability attacks which are endemic to JavaScript. For example, key interfaces like `Math.random()` and `XMLHttpRequest` can simply be overwritten with application-defined code. Such dynamic patching is not necessarily a sign of evil—for example, diagnostic tools like Mugshot [23] use such patching to debug application state. However, if a frame contains two JavaScript libraries from domains `X` and `Y`, and `X`'s code loads first, then `Y` cannot guarantee that its code has unmediated access to the real system interfaces.

Dangerous built-in statements: JavaScript provides several constructs for dynamically compiling and executing source code; examples include the `eval()` method and the `Function()` constructor. Malicious applications can use these interfaces to hide attack code from static security analyses. Prohibiting access to these interfaces is tricky because JavaScript defines many aliases for such functions, e.g., `window.eval()`, `window.parent.eval()`, `window.top.eval()`, etc. [19].

JavaScript also provides statements like `with` and `delete` that allow programs to dynamically modify the scope chain that the runtime consults to resolve variable names. Using these statements, attackers can circumvent dynamic security mechanisms used by naïve code rewriters (see Figure 4).

Complex "this" semantics: As implied by the previous paragraph, the built-in `window` object is an alias for the global namespace. For example, the global `XMLHttpRequest` object can also be referenced via `window.XMLHttpRequest`. If a secure application wants to prevent an untrusted library from accessing sensitive globals, the application must prevent that library from accessing `window`. Unfortunately, doing so is tricky, since JavaScript provides multiple ways to access `window`. For example, all JavaScript functions are actually methods, and if a method is called unbound, e.g., as `f()` instead of `someObj.f()`, the method's `this` reference is automatically set to `window`. The browser also uses `window` as the `this` reference for timer callback functions and other event handling callbacks. Source code that is dynamically evaluated via `Function()` will also use `window` as `this`.

B. Isolation Using Iframes

Each frame represents a separate JavaScript runtime. If two frames belong to the same origin[1], they can manipulate each other's JavaScript state using mechanisms like the `window.frames` array. If two frames belong to *different* origins, cross-frame interactions are restricted to the `postMessage()` API, which asynchronously transfers immutable strings.

Isolation frameworks like Privilege-separated JavaScript (PSJ) [3] place untrusted libraries in separate iframes. A trusted master frame implements sensitive functionality, and exposes that functionality to untrusted frames via asynchronous pass-by-value RPCs that are layered atop `postMessage()`. The master frame also routes messages between untrusted frames.

When the master frame creates an untrusted frame, the master initializes the new frame in some way, e.g., by deleting references to sensitive functions, or creating virtualized copies of sensitive functions (see §VI for more detail). The master also injects a communication library which layers a high-level RPC protocol atop raw `postMessage()` calls. The master then loads the untrusted code, which uses the communication library to interact with the master.

Advantages: If the master trusts *nothing* in an untrusted frame (including the communication stub that the master injected at initialization time!), the master can rely on the browser to isolate the master and the untrusted frames. This is because each frame represents a separate JavaScript runtime that can only exchange immutable strings with other frames. Thus, if an untrusted frame falls victim to prototype poisonings, mutability attacks, or dangerous JavaScript statements, other frames (included the trusted master) are still safe.

Disadvantages: Frame-based isolation forces RPCs to be asynchronous. Asynchronous semantics are often well-suited for IO operations, and JavaScript uses asynchronous callbacks to handle user input and network operations (§III-B). Unfortunately, many types of mashup communication are best expressed using *synchronous* RPCs. For example, suppose that the master wants to incorporate a cryptography library that defines a hash function. The master's code can be written in a much simpler way if the call to the hash function is synchronous. If cross-domain interactions must be asynchronous, then any master function f which invokes an RPC must be split into a "top half" that initiates the RPC, and a "bottom half" callback that asynchronously receives and processes the result. This asynchronous refactoring affects any code in the master that calls f—since f is now asynchronous, anything that invokes f must define a callback which is fired when f has completed. Even worse, if f's bottom half must invoke additional RPCs, then this bottom half must be split too. This stack splitting is difficult for developers to manage across long, multi-hop asynchronous call chains [2]; as a result, there are many projects which attempt to introduce more synchrony into JavaScript [25], [26], [27], [36] (although all of these projects provide pseudo-synchrony, not Pivot's true synchrony (§VI)).

[1] An origin is defined as a three tuple <protocol, host, port>, e.g., <https, cnn.com, 80>.

C. Isolation Using Code Rewriting

Isolation systems like Caja [24] and Jigsaw [22] place each domain's code inside the same frame. To make this co-location safe, the isolation system must rewrite each domain's code. For example, dangerous statements like `delete` are statically removed at rewrite time. Preventing other security problems requires the insertion of runtime checks. For example, JavaScript allows object properties to be accessed using dot notation (`obj.x`) and bracket notation (`obj["x"]`). Bracket notation allows the property specifier to be an arbitrary expression, like the return value of a function (`obj[f()]`). To prevent manipulation of the `__proto__` property, the rewriter must instrument property accesses that use the bracket notation.

As an application executes, it generates call chains which may cross isolation boundaries. For example, a library from domain `X` may invoke code from domain `Y`; in turn, `Y`'s code may invoke a sensitive browser function. Each library has distinct security privileges which constrain the allowable interactions. To enforce these privileges, the isolation system must rewrite functions so that they track execution context. For example, Jigsaw maintains a privilege stack which represents the access rights of the corresponding functions on the call stack; by examining the top of this stack, the Jigsaw runtime can determine the access privileges of the currently executing function. To implement this privilege stack, Jigsaw rewrites function definitions so that, on function entry, a new entry is pushed onto this stack, and on function return, that entry is popped.

Advantages: Since all libraries reside within the same frame, they can interact using synchronous RPCs. As discussed earlier, synchronous RPCs simplify cross-domain interactions and make applications easier to understand. Rewriting systems also allow libraries to exchange data by reference instead of by value (as `postMessage()` requires). Pass-by-reference is safe because the security runtime sanitizes objects as they cross isolation boundaries, wrapping the objects in proxies that only reveal sensitive properties to authorized execution contexts. Using pass-by-reference, applications can avoid complex marshaling operations that `postMessage()`-based sharing may require.

Disadvantages: Compared to frame-based systems, rewriting solutions impose high performance costs. Whereas iframe isolation is implemented by fast C++ code inside the browser, rewriting systems use application-level checks that are implemented in JavaScript. Depending on the application, these checks can reduce performance by 10x or more [12], [22], [31]. Since multiple, untrusted libraries are forced to share the single DOM belonging to a single frame, the rewriter must expose a virtualized DOM interface to each library. The security checks in this virtualized DOM further reduce performance. Rewriting systems also force developers to write code in a new JavaScript dialect which has non-standard semantics as a result of restricted operations and forbidden (but occasionally useful) statements like `eval()`.

```
<html>
<head>
  <title>A Pivot app</title>
  <script src="Pivot-master.js"></script>
</head>
<body>
  <script>
      var satX = {url: "http://x.com/x.html",
                  namespace: "x"};
      var satY = {url: "http://y.com/y.html",
                  namespace: "y"};
      var urlsToLoad = [satX, satY];
      Pivot.createSatellites(urlsToLoad);

      //Additional trusted master code . . .
  </script>
</body>
</html>
```

Figure 5: The master frame of a simple Pivot application.

D. Summary

Historically, iframe isolation has provided high performance, but it has required asynchronous, pass-by-value RPCs. Rewriting systems provide synchronous pass-by-reference RPCs, but performance is poor, and developers cannot use raw JavaScript. Ideally, developers could use the full, unconstrained JavaScript language to write RPCs that are synchronous, high-performance, and capable of using pass-by-reference arguments. In the next section, we describe how Pivot achieves three of the four goals, providing synchronous, fast mashups that are written in raw JavaScript. We describe how the unmet goal (pass-by-reference RPC arguments) is difficult to implement efficiently.

III. Design

At initialization time, a Pivot application consists of a single frame (see Figure 5). This frame contains three items: the trusted JavaScript code belonging to the Pivot master library; the trusted, application-specific JavaScript code which implements sensitive operations; and a list of URLs representing untrusted content to load in the satellite frames. Pivot's master library is responsible for creating new satellites and routing RPC messages between the satellites.

Each satellite frame contains rewritten (but untrusted) application code, and an untrusted Pivot scheduler. A satellite's generator function (§III-A) yields to its local scheduler upon invoking an RPC. The scheduler sends the RPC request to the master frame, receives the response from the master frame, and restarts the local generator function. The scheduler also maintains the queue of deferred asynchronous event handlers. Using this queue, the scheduler maintains the illusion of single-threaded execution within a frame (§III-B). Figure 6 provides an example of a satellite frame.

```
<html>
<head>
  <title>A satellite frame</title>
  <script src="Pivot-satellite.js"></script>
</head>
<body>
  <script>
      //Declare some functions . . .
      function md5(data){...};
      function sha1(data){...};

      //. . . and make those functions
      //externally visible as RPC names.
      Pivot.registerRPC("md5", md5);
      Pivot.registerRPC("sha1", sha1);

      //Invoke an RPC declared by an
      //external compression library.
      var gzip = Pivot.getFunction("compression",
                                   "gzip");

      //Invoke the RPC synchronously, just
      //like a local function.
      var compressedStr = gzip("abcdefg . . .");
  </script>
</body>
</html>
```

Figure 6: A satellite frame in a simple Pivot application. This figure depicts the *unrewritten* version of the frame, i.e., this is the code that the developer writes. See Section III-A for a description of how Pivot rewrites code.

A. Rewriting: Creating Generators

Each JavaScript frame represents a single-threaded execution context. When a frame loads, it executes top-level code that is equivalent to the `main()` function of a C program. Once the top-level code finishes, the browser can fire application-defined event handlers in response to asynchronous events like user inputs or network activity. In this section, we ignore asynchronous event handlers and describe how to transform the top-level code into a generator. In Section III-B, we discuss how Pivot supports asynchronous event handlers.

Figure 7(a) provides a simple example of RPC invocation. From the developer's perspective, the RPC invocation looks like a normal JavaScript function call. However, Pivot rewrites the RPC call and adds bookkeeping code to coordinate the behavior of the satellite frame and the master frame.

Creating the generator function: First, Pivot wraps the top-level application code in a function called `program()` (see Figure 7(b)). Pivot replaces the RPC import statement (`Pivot.getFunction()`) with code that defines a generator function. This generator represents a client-side RPC stub. When invoked, the stub uses `postMessage()` to send an RPC request to the master frame. The stub then yields the special value `Pivot.RPC_YIELD`. Later, when the generator resumes execution, it assumes that the local Pivot library has

```
var sha1 = Pivot.getFunction("crypto",
                             "sha1");
var retVal = sha1("foo");
```

(a) The unrewritten code invokes an RPC and stores the result.

```
//Pivot wraps the entire application in a
//generator function called "program".
var program = function(){
  //Pivot rewrites the sha1 RPC stub
  //to send a postMessage() request
  //to the master frame, then yield.
  var sha1 = function(){
     postMessage(window.parent, //The master frame.
         {namespace: "crypto",
          funcName:  "sha1",
          args: Pivot.serialize(arguments)});
            //The "arguments" keyword is a
            //predefined JavaScript array
            //containing a function's arguments.
     yield Pivot.RPC_YIELD;

     //When control flow reaches here, it
     //means that Pivot has restarted the
     //yielding RPC stub, placing the RPC
     //return value in Pivot.RPCRetVal.
     yield Pivot.deserialize(Pivot.RPCRetVal);
  };

  //The next six lines are the rewritten
  //version of "var retval = sha1('foo')".
  var __tmp__, __gen__ = sha1("foo");
  while((__tmp__ = __gen__.next()) ==
                          Pivot.RPC_YIELD){
     yield Pivot.RPC_YIELD;
  }
  var retVal = __tmp__;

  yield Pivot.CALL_CHAIN_FINISHED;
};
program = program(); //Get the actual generator.
```

(b) The rewritten code is a generator that yields upon RPC invocation.

Figure 7: Transforming an application into a generator function.

received the RPC response and placed the response in the special value `Pivot.RPCRetVal`. The stub simply returns the value to the local caller of the stub.

Pivot rewrites each function call site to check for the special return value `Pivot.RPC_YIELD`. If a function returns this value, it means that the function (or something in its downstream call chain) invoked an RPC. If a call site receives a `Pivot.RPC_YIELD` value, that call site will yield that value as well. This yielding occurs all the way up the call stack, creating a chain of paused generator functions that is rooted by the `program()` function. Note that, although Pivot rewrites every function invocation, it only has to unwind and rewind call stacks if an RPC is invoked.

The local scheduler: Figure 8 depicts `Pivot.run()`. This function issues the initial call to `program()`, and it is the function to which `program()` yields. If `program()` returns `Pivot.RPC_YIELD`, then `Pivot.run()` sim-

```
Pivot.run = function(){
    var retVal = program.next();
    switch(retVal){
        case Pivot.RPC_YIELD:
            //Need to wait for the master
            //to send an RPC response via
            //postMessage() . . .
            return;
         case Pivot.CALL_CHAIN_FINISHED:
            //Do we need to run deferred
            //handlers for asynchronous
            //events?
            Pivot.drainQueue();
            return;
         default:
            alert("Error!");
            break;
     }
}
```

Figure 8: The Pivot scheduler simply invokes a generator function (Figure 7(b)) and checks whether it yielded due to an RPC invocation.

```
//Handle incoming postMessage()s.
Pivot.handlePM = function(evt){
   function(evt){
      switch(evt.data.type){
        case Pivot.RPC_RESPONSE:
           //Local code invoked an RPC, and we
           //have received the response. Resume
           //execution of local RPC stub!
           Pivot.RPCRetVal = evt.data.RPCData;
           Pivot.run();
           break;
        case Pivot.RPC_REQUEST:
           //External satellite wants to call
           //a locally defined function.
           var retVal = Pivot.callLocalFunc(evt);
           var resp = {type: Pivot.RPC_RESPONSE,
                       RPCData: retVal};
           evt.source.postMessage(resp,
                                  window.parent);
           break;
        default:
           alert(''Error!'');
     }

};
window.addEventListener("message",
                    Pivot.handlePM);

//Start the rewritten application!
Pivot.run();
```

Figure 9: At the bottom of a satellite frame, Pivot places code to respond to `postMessage()`s and to start the frame's generator function.

ply terminates, and its satellite frame waits for an RPC response via `postMessage()`. If `program()` returns `Pivot.CALL_CHAIN_FINISHED`, then `Pivot.run()` knows that `program()` has terminated naturally, i.e., it has no more top-level code to run (see the bottom of Figure 7(b)). In this case, `Pivot.run()` knows that it is now safe to execute any deferred event handlers. We discuss how Pivot processes these handlers in Section III-B.

RPC responses: Figure 9 shows the `postMessage()` handler that Pivot inserts into each satellite. When the satellite receives an RPC response, Pivot extracts the RPC result from the `postMessage()` data, stores it in `Pivot.RPCRetVal`, and invokes `Pivot.run()`. `Pivot.run()` invokes the paused `program()` routine. `program()` resumes execution at the top of the while-loop which invokes the `sha1()` RPC stub (see Figure 7(b)). When the stub resumes, it finds the RPC result in `Pivot.RPCRetVal`. The stub returns that value to `program()`. `program()` consumes the value and then terminates by yielding the special `Pivot.CALL_CHAIN_FINISHED` value. This signals to `Pivot.run()` that the program's top-level code is done.

Summary: Figures 7(b), 8, and 9 collectively represent the rewritten version of the simple application in Figure 7(a). Figure 10 depicts the end-to-end RPC control flow.

B. Deferring Asynchronous Events

From the application's perspective, a standard frame is single-threaded and non-preemptive—at any given moment, at most one application-defined call chain is executing. An application can define callbacks which fire in response to asynchronous events, but a callback's execution can never overlap with the execution of another callback or the application's top-level code.

As currently described, Pivot RPCs can violate these concurrency semantics. This is because the browser's JavaScript scheduler is unaware of Pivot's application-level scheduler. For example, suppose that a satellite frame defines a callback handler for mouse clicks. If the frame makes an RPC, the frame's generator chain will yield to Pivot, and Pivot will consider the frame to be paused. However, while the RPC is in-flight, the user may click on a visual element within the frame, causing the browser to fire the mouse callback. The callback will fire while the RPC is still in-flight, violating developer expectations that Pivot RPCs act like regular, non-yielding function calls.

To preserve the expected concurrency semantics, Pivot rewrites asynchronous event handlers so that they are scheduled by Pivot instead of the browser. Pivot's scheduler then ensures that each satellite has only one active call chain at any given time. Whereas Pivot uses static rewriting to instrument function call sites, it uses dynamic patching of the JavaScript runtime [21], [23] to help the static rewriter enforce single-threaded semantics. Pivot redefines the registration interfaces for asynchronous event handlers, interposing on timer interfaces like `window.setTimeout()`, GUI interfaces like `DOMNode.addEventListener()`, and network interfaces like `XMLHttpRequest.addEventListener()`. The modified registration interfaces simply wrap each handler in code that adds the real handler to Pivot's deferred execution queue.

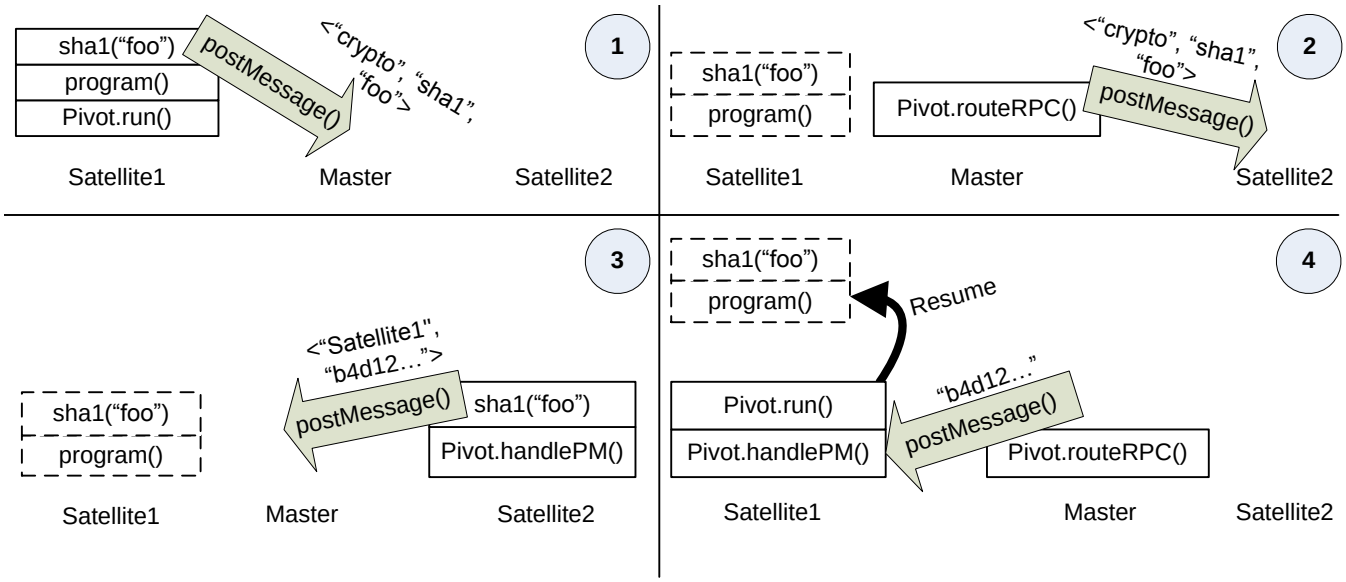

Figure 10: The control flow of the RPC from the running example.
Step 1: The satellite frame begins to execute. `Pivot.run()` invokes the satellite's generator function `program()` (see Figure 7(b)); in turn, this generator invokes the RPC stub `sha1()`. The stub issues a `postMessage()` to the master frame and then yields control. Upon detecting that `sha1()` has yielded, `program()` yields too. `Pivot.run()` then terminates.
Step 2: Satellite frame 1 saves the generator chain belonging to `program()` and `sha1()`. Meanwhile, the master frame handles the incoming `postMessage()`, routing it to satellite frame 2, which implements RPCs belonging to the "crypto" namespace.
Step 3: Satellite frame 2 receives the RPC request, invokes the `sha1()` function, and returns the result to the master frame, tagging the result with the name of the destination frame.
Step 4: The master frame delivers the RPC response to satellite frame 1. The `postMessage()` handler invokes `Pivot.run()`, which invokes the paused `program()` function; in turn, `program()` invokes the yielded `sha1()` stub, which returns the RPC result to the call site in `program()`.

```
Pivot.__setTimeout__ = window.setTimeout;
window.setTimeout = function(rewrittenCb, ms){
   var wrappedCb = function(){
      Pivot.deferredQ.append(rewrittenCb);
   };
   Pivot.__setTimeout__(wrappedCb, ms);
};
```

Figure 11: Dynamically interposing on the browser's interface for registering timer callbacks.

The JavaScript language has powerful reflection capabilities, and to a first approximation, all JavaScript objects are mutable, enumerable dictionaries. Thus, to interpose on an event registration interface, Pivot simply assigns Pivot-defined code to the relevant object property, as shown in Figure 11. In practice, JavaScript's reflection interfaces have subtle semantics, and these interfaces are implemented in semi-incompatible ways across different browsers. Thus, Figure 11 provides a simplified description of Pivot's dynamic patching. We defer a detailed description of JavaScript reflection semantics to other work [21], [23].

So, whenever the browser tries to execute an asynchronous handler, Pivot just adds that handler to a Pivot-controlled queue. To maintain the illusion of single-threaded execution, Pivot only drains this queue when a frame has no in-flight RPCs. Pivot detects this condition inside the local scheduler (Figure 8). When a generator chain yields `Pivot.CALL_CHAIN_FINISHED`, Pivot is guaranteed that there are no active generator chains inside that frame. Thus, `Pivot.drainQueue()` can safely iterate through the list of deferred callbacks and execute each one.

Pivot rewrites all function call sites, including those inside event handlers. Thus, deferred event handlers can invoke RPCs. As shown in Figure 12, `Pivot.drainQueue()` detects when a callback `cb` returns `Pivot.RPC_YIELD`. When this happens, `Pivot.drainQueue()` sets the global variable `program` to `cb`, and then immediately terminates execution. Pivot's standard RPC mechanisms will then ensure that the RPC is completed, and that the rest of the deferred handlers are eventually executed.

Pivot executes callbacks in FIFO order to prevent starvation of early-arriving callbacks. If a callback executes a

```
Pivot.drainQueue = function(){
   while(Pivot.deferredQ.length > 0){
      var cb = Pivot.deferredQ.pop();
      var retVal = cb.next();
      switch(retVal){
         case Pivot.RPC_YIELD:
            //The callback made an RPC!
            //We cannot issue any more
            //deferred handlers until
            //this RPC completes . . .
            program = cb;
            return;
         case Pivot.CALL_CHAIN_FINISHED:
            //The deferred handler did
            //not issue an RPC, so it's
            //safe to execute another one.
            break;
         default:
            alert("Error!");
            break;
      }
   }
};
```

Figure 12: Executing deferred asynchronous callbacks.

long-running Pivot RPC, other callbacks will stall. This is no different than the impact of a slow local function call in a single-frame system like Jigsaw. However, Pivot uses RPC timeouts to keep a buggy or malicious satellite from indefinitely blocking a caller frame (§III-D).

C. Sandboxed Iframes

By default, iframes from the same origin can interact with each other's JavaScript state using references like `window.parent`. If a developer wishes to self-host multiple untrusted libraries, the developer should use the best practice of serving each library from a separate origin. Alternatively, Pivot can automatically place each library into a "sandboxed" iframe [35] which we describe below.

The new HTML5 standard extends the `<iframe>` tag with an optional `sandbox` attribute. By default, sandboxed frames can only display visual content—they cannot run plugins, execute JavaScript code, or include forms. By specifying parameters for the `sandbox` tag, e.g., `<iframe sandbox="allow-scripts">`, the creator of the frame can selectively enable permissions. However, if the sandboxed iframe does not have the `allow-same-origin` permission, the browser gives the frame a unique, random origin that is only known to the browser. The browser uses this unique origin, not the true one, when applying the same-origin policy to the sandboxed frame. So, by default, JavaScript in the sandboxed frame cannot access content belonging to any real origin. For example, the frame will be unable to access cookies or other client-side storage belonging to its true origin; the frame will also be unable to send `XMLHttpRequests` to its home server. Even if the sandboxed frame has the same true origin as its parent frame, it cannot use JavaScript references like `window.parent` to inspect the JavaScript state of its parent. Sandboxed frames are allowed to communicate with their parent frames using `postMessage()`, and they can display visual information if their parent frames have given them a non-zero sized viewport.

Pivot can optionally place each untrusted library into a sandboxed frame that lacks the `allow-same-origin` permission. By doing so, Pivot frees the developer from the burden of maintaining separate origins for each self-hosted library. Furthermore, if a master frame gives a sandboxed satellite frame a zero-sized viewport, the satellite is restricted to performing pure computation. All accesses to the network, the visual display, user input, and local storage must be requested through `postMessage()`, such that the master frame vettes each request and executes the authorized ones on behalf of the untrusted satellite.

D. Discussion

The JavaScript runtime provides built-in functions like `Math.random()` and `Date()`. These functions never return `Pivot.RPC_YIELD`. Thus, Pivot correctly treats these functions like application-defined functions that are not RPCs (although Pivot does need a call-site check not shown in Figure 7(b) which tests whether a function is an application-defined generator or a built-in function). A few built-in methods internally call an application-defined function; for example, `Array.sort()` uses such a function to compare array elements. Pivot disallows such functions from invoking RPCs, since Pivot cannot force C++ code inside the browser to yield to Pivot's application-level scheduler.

Figure 7(b) provides a simple example of how Pivot rewrites a function call that is the right-hand side for an assignment operation. Function calls can arise in a variety of additional contexts—a function's return value can be the test condition for an if-statement, the argument to another function, and so on. For a given function call, Pivot's general rewriting strategy is to find the lowest ancestor in the abstract syntax tree that is a block statement. The hoisted, rewritten call which checks for `Pivot.RPC_YIELD` is inserted as a new AST child for the block; this child immediately precedes the one which contained the original function call. Pivot then replaces the original function call with the temporary value set by the hoisted, rewritten call.

A buggy or malicious satellite that is servicing an RPC may hang, i.e., the satellite may never return a value to the Pivot scheduler. To prevent a denial-of-service attack on the RPC initiator, Pivot will timeout an RPC that is taking too long to complete, forcing the RPC to return an error value to the initiator. The Pivot scheduler (Figure 8) implements these semantics by setting a JavaScript timer when a generator yields the value `Pivot.RPC_YIELD` (the actual timer code is not shown in the simplified figure). When the Pivot `postMessage()` handler receives an RPC response (Figure 9), Pivot cancels the relevant timer. If the timer is

not canceled, it will eventually restart the satellite frame which initiated the hung RPC, setting `Pivot.RPCRetVal` to an error value.

Pivot relies on the browser to enforce memory isolation between the trusted master frame and the untrusted satellite frames. However, nothing prevents a satellite from trying to subvert the Pivot infrastructure within its own frame. For example, a satellite can directly generate RPC requests by crafting its own `postMessage()` calls. A satellite can also try to attack Pivot's virtualized event framework, e.g., by looking for baroque JavaScript aliases to the underlying non-virtualized functions [19]. Such chicanery does not affect end-to-end application security because Pivot does not trust anything in a satellite, including the satellite's local Pivot code. The master frame verifies that all RPC messages are well-formed, destined for extant frames, and sent from domains which are allowed to contact the specified destinations. Thus, if a satellite directly invokes `postMessage()`, it can only invoke RPCs that the trusted master frame already allows. The `postMessage()` interface only exchanges immutable strings, so it cannot be used to tamper with master frame objects by reference. If a satellite disrupts Pivot's local scheduling framework, the satellite may disrupt its own single-threaded execution semantics, but other frames will not be affected.

Ideally, a mashup isolation system would provide synchronous RPC semantics with pass-by-reference sharing. Pivot provides the former by interposing on function calls and translating cross-domain calls into generator yields. However, Pivot uses `postMessage()` as its cross-domain transport layer; since `postMessage()` exchanges immutable strings, Pivot provides pass-by-value cross-domain sharing. One could emulate pass-by-reference by interposing on object property accesses, and translating each write to an RPC that is reflected to the object's "home" frame. However, it seems difficult to make such a scheme fast.

IV. Implementation

Our client-side Pivot system consists of three libraries: `Pivot-satellite.js`, `Pivot-master.js`, and `pmLib.js`, which implements a `postMessage()` RPC protocol that is used by the first two libraries. The three libraries are 21 KB in total. To rewrite code, we use an ANTLR [28] toolchain similar to the one used by Jigsaw. The ANTLR parser translates JavaScript libraries into ASTs; a custom Pivot rewriter modifies those libraries as described in Section III and then converts the ASTs back to JavaScript.

V. Evaluation

In this section, we compare Pivot to Jigsaw [22], a state-of-the-art mashup framework that places all JavaScript libraries in the same frame. Our evaluation demonstrates three major points:

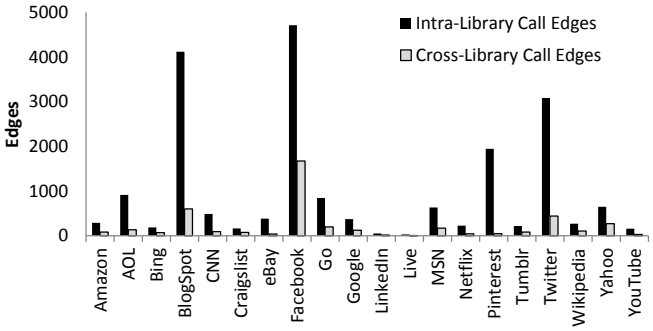

Figure 13: Call graph edges in Alexa's top 20 sites for the US. A single edge represents one or more invocations of a unique caller/callee pair.

- Using an empirical analysis of JavaScript call graphs in real web applications, we demonstrate that the vast majority of function calls do not cross library boundaries. Thus, Pivot's trusted master/untrusted satellite decomposition is a natural one (§V-A).
- Since Pivot places untrusted libraries in separate frames, it does not need to rewrite those libraries if they do not make synchronous RPC calls. In contrast, Jigsaw must rewrite all libraries. By avoiding many of the dynamic security checks associated with full writing, Pivot RPCs are typically one to two orders of magnitude faster than Jigsaw RPCs (§V-B). Overall, a Pivot version of the Silo web application [20] is 12.6 times faster than an equivalent Jigsaw version.
- Firefox's current implementation of `yield` is slow (§V-C), so rewritten Pivot code is currently slower than rewritten Jigsaw code (§V-E). However, Pivot rewrites less code than Jigsaw, so overall, Pivot applications will be faster.

We ran all experiments on a machine with two 2.67 GHz processors and 4 GB of RAM. All web pages were loaded in Firefox v18.0.2. Firefox is currently the only browser to implement the `yield` keyword; however, this keyword is an official part of the upcoming JavaScript v6 specification [9], and other major browsers will soon implement the keyword. For example, Chrome already provides the feature in developer builds of the browser (although the implementation still has some bugs, which is why we do not present performance graphs for Pivot on Chrome).

A. Cross-library Call Graphs

To determine how often web pages make cross-library function calls, we visited the top 20 websites in the United States as determined by Alexa. We interacted with those pages as a normal user might, e.g., by scrolling down to force the page to dynamically load below-the-fold images. As we interacted with each page, we used Firefox's built-in JavaScript debugger to capture a call tree for each site.

Amazon

Function	% cross-lib callers	% cross-lib callees
<anonymous>(*,*)	5.37%	0%
init(*, *, *)	0%	0%
c.Event(*, *)	0%	0%
H()	0%	0%
type(*)	0%	0%

CNN

Function	% cross-lib callers	% cross-lib callees
this._methodized()	5.69%	0%
$A(*)	0%	0%
Object.extend(*, *)	1.22%	0%
D(*)	0%	0%
extend(*, *)	0%	0%

Figure 14: The top five most invoked functions on Amazon.com and CNN.com: how often those functions were called by external libraries, and how often those functions called functions defined by external libraries. We omit results for other sites due to space, but other sites had similar or lower numbers of cross-library edges.

As Figure 13 shows, the vast majority of caller/callee edges do not straddle library boundaries. Firefox's debugger only records unique caller/callee edges, not the number of times a page traverses those edges, so Figure 13 still admits the possibility that the small number of cross-library edges are heavily traversed. To investigate this hypothesis, we used a DynaTrace analytics web proxy [6] to instrument each JavaScript function in each page and collect per-edge traversal statistics. We found that cross-library calls are exceedingly rare, as shown in Figure 14. Furthermore, the cross-library edges almost always involved a caller residing in core application "glue" code, and a callee residing in a library like jQuery which is devoted to one task and which rarely makes outcalls to the core application. Thus, Pivot's architecture (a small, trusted master and multiple untrusted satellites) is a natural fit for how developers already design web pages.

B. End-to-end RPC Latency

To test the end-to-end latencies of cross-domain RPCs, we built a mashup application that integrated three untrusted JavaScript libraries. We picked these particular libraries because they were used to evaluate Jigsaw in the original Jigsaw paper [22]. Our mashup application, called Shard, is a privilege-separated implementation of the Silo application [20]. The client-side Shard JavaScript collaborates with a web server to layer a delta-encoding protocol atop HTTP. When a user needs to fetch an HTML, JavaScript, or CSS object, Shard downloads that object using an AJAX connection. Shard then splits that object into chunks, calculates

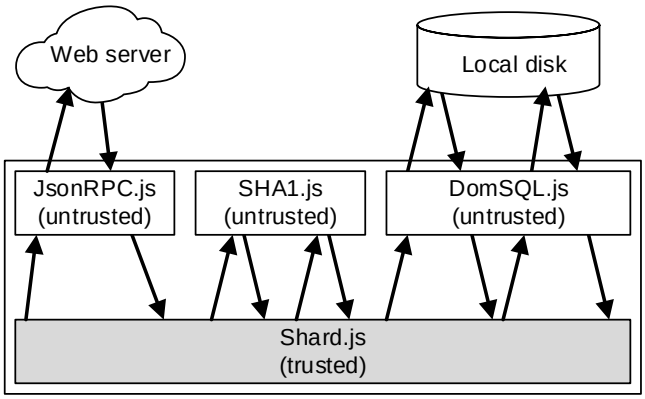

Figure 15: Architecture of the Shard mashup: fetching data from a web server, calculating the hash of the chunks, and storing those chunks on the local disk.

a hash for each chunk, and stores each $< hash, chunk >$ pair in local DOM storage [33]. Later, if the user needs to fetch the latest version of that object, Shard sends a list of locally resident chunk ids to the server. If the object has changed, the server only sends the new object chunks, avoiding the cost of sending the entire object. Once Shard has received the new chunks, it fetches the relevant old chunks from DOM storage, reconstructs the new object, and uses `document.write()` to insert the new object into the page's HTML.

As shown in Figure 15, Shard fetches objects using the JsonRPC AJAX library [11]. Shard uses the Stanford JavaScript Crypto Library [30] to calculate SHA1 chunk ids, and Shard writes those chunks to DOM storage using DomSQL [5], a JavaScript library which layers a SQL query interface atop DOM storage.

We built two versions of Shard, one that used Pivot, and another that used Jigsaw. The Pivot version placed unrewritten library code in each satellite frame; the code in the trusted master frame was rewritten to allow the master to make synchronous RPCs to satellite-defined functions. In the Jigsaw version of Shard, each library had to be rewritten by the Jigsaw compiler; this ensured that it was safe to place mutually untrusted libraries within the same frame. As a performance baseline, we also implemented a version of Shard which used standard JavaScript and which included the unrewritten libraries in the same frame.

The first three results in Figure 16 compare the performance of Shard RPCs in Jigsaw and Pivot. Results are normalized with respect to the performance of the baseline implementation. Note that, for the JsonRPC calls, Shard contacted a localhost web server. This allowed us to minimize networking costs and focus on the CPU overhead of the isolation frameworks.

For RPCs to the SHA1 library and DomSQL, Pivot was 20.6x faster and 128.2x faster than Jigsaw. Because Jigsaw

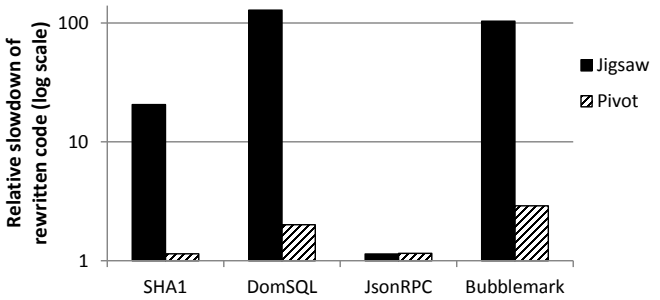

Figure 16: Normalized, end-to-end RPC latencies: Jigsaw versus Pivot. RPC latency is defined as the time that elapses between the invocation of an RPC and the reception of the RPC result. Thus, RPC latency includes both message transfer costs, and the computational costs of handling the RPC request (e.g., issuing a DomSQL operation).

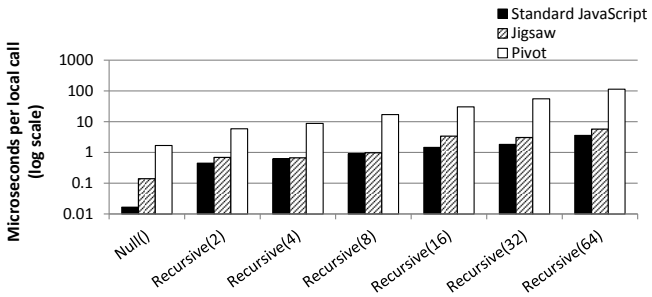

Figure 17: Invocation latencies for a local null function and recursive local null functions, demonstrating the impact of generator overhead.

placed all libraries in the same frame, it had to inject costly dynamic security checks into each library. Pivot avoided those costs by placing unrewritten libraries in separate frames. Jigsaw also had to virtualize the DOM interface, presenting each library with a DOM emulation layer that performed security checks before possibly accessing the real DOM. Thus, when a Jigsaw library tried to access DOM storage, it had to interact with an additional software layer that Pivot applications did not have to traverse (each Pivot satellite lived in its own frame and had private, isolated DOM state).

Jigsaw virtualizes the `XMLHttpRequest` object, but Pivot and Jigsaw had similar performance for JsonRPC calls (1.15x slowdown versus 1.14x slowdown). This is because, even though JsonRPC contacted a localhost server, the end-to-end latency of the RPC was still dominated by HTTP transfer costs.

The final result in Figure 16 shows the performance of a simple mashup application that invoked RPCs on Bubblemark [10], a DOM-intensive animation program. The trusted application core issued commands to the Bubblemark library like "start animation," "stop animation," and "increase the number of animated objects." Pivot is much faster than Jigsaw for Bubblemark RPCs, just like it is faster for DomSQL RPCs, because Pivot does not incur DOM virtualization costs.

C. Generator Overhead

If a library does not need to issue synchronous RPC calls, then Pivot can place the unrewritten library in an iframe and avoid the rewriting costs incurred by Jigsaw. However, when rewriting is necessary (e.g., in a master frame), Pivot leverages the JavaScript `yield` statement to implement generator functions. Firefox's current implementation of generators is very slow. Calling an unrewritten null function required 0.03 microseconds. However, creating a new generator took an order of magnitude longer (0.39 microseconds), and calling `next()` on a null generator was similarly slow (0.44 microseconds). We also found that accessing an object property within a generator was 22 times slower than accessing an object property within a regular function. There is no fundamental reason why generator overhead must be this high, since there is prior work from the programming languages community which describes how to make generators fast [16]. Thus, we expect generator performance to improve once the new JavaScript specification is implemented in more browsers and there is more developer pressure to make `yield` fast. Until that happens, Pivot's ability to safely integrate unrewritten code is crucial for performance.

D. End-to-end Performance Improvement

To compare the end-to-end performance of our Shard implementations, we measured the total time that an implementation needed to fetch a 50 KB web object, split the object into chunks, and store those chunks in DOM storage. This end-to-end time captured both RPC latencies and the computation time within the trusted application core and the untrusted libraries. Overall, Shard on Pivot was 12.6 times faster than Shard on Jigsaw. Even though Pivot RPCs can be 100 times faster than Jigsaw RPCs (Figure 16), generator overhead within the rewritten master Shard frame (Figure 17) can make rewritten Pivot code slower than rewritten Jigsaw code. Nonetheless, end-to-end, Pivot is still an order of magnitude faster than Jigsaw due to faster RPCs and Pivot's ability to selectively rewrite code. As `yield` implementations get faster, we expect Pivot's performance advantage to grow.

E. RPC Communication Overheads

Figure 18 compares Jigsaw and Pivot, showing the end-to-end latencies for a null RPC that immediately returns, and a recursive null RPC that calls itself n times before returning null. These tests isolate the raw cost of RPC invocation, ignoring any computation that an RPC library might do. Jigsaw's function invocation overheads are 3–5 microseconds across all tests. Pivot's overheads are slower by two

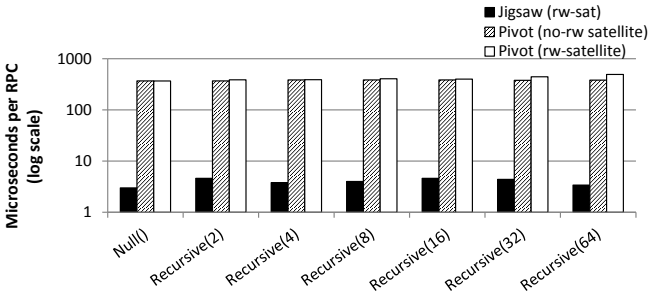

Figure 18: End-to-end RPC latencies for a null RPC and recursive null RPCs: Jigsaw, Pivot with an unrewritten satellite, and Pivot with a rewritten satellite.

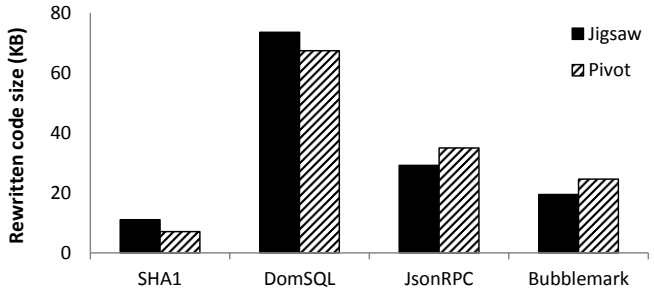

Figure 19: Relative expansion of rewritten code: Jigsaw versus Pivot.

orders of magnitude, ranging from 369–491 microseconds. There are several sources for this overhead. First, each Pivot RPC requires two `postMessage()` roundtrips (one between the calling satellite and the master, and another between the master and the responding satellite). These RTTs add roughly 230 microseconds to each Pivot RPC. Second, as explained in Section V-C, generator overhead grows as the depth of the call chain increases. This explains why, in Figure 18, the white bar slowly grows compared to the striped bar.

An asynchronous `postMessage()` framework like PostMash [4] incurs the `postMessage()` RTT costs described above; however, such a framework avoids the additional generator latencies that Pivot introduces. If a particular mashup application only requires asynchronous cross-domain communication, then Pivot's generators add unnecessary overhead, and a framework like PostMash is more appropriate. However, for applications that *do* require synchronous RPCs, Pivot can provide order-of-magnitude performance improvements over other synchronous frameworks (§V-D).

F. Size of Rewritten Code

Figure 19 depicts the size of fully rewritten libraries in Jigsaw and Pivot. Pivot's rewritten libraries are similar in size to those of Jigsaw, so Pivot does not fundamentally change the amount of code that a secure web application must download. However, Pivot has the ability to safely integrate unrewritten libraries as well, and such libraries are typically 2–4 times smaller than the rewritten versions.

VI. Related Work

Leveraging browser-provided isolation: TreeHouse [14] and Privilege-separated JavaScript (PSJ) [3] use browser-enforced protection domains. PSJ employs frames, and TreeHouse uses web workers. Web workers are similar to frames; each worker has a separate JavaScript runtime, and communicates with other workers and frames using `postMessage()`. The primary difference between a frame and a web worker is that the latter does not have a DOM tree.

Pivot, TreeHouse, and PSJ all use a trusted coordinator frame to route messages and determine which sensitive operations can be invoked by untrusted code. All three systems also place a control stub in each untrusted frame. However, Pivot and PSJ do not trust those control stubs, whereas TreeHouse does. Since TreeHouse does not rewrite untrusted code, malicious libraries can use attacks like prototype poisoning (§II-A) to tamper with TreeHouse's control stubs.

Neither TreeHouse nor PSJ provide synchronous RPC interfaces. Thus, porting legacy applications to TreeHouse and PSJ will often require non-trivial refactoring to convert synchronous code paths to asynchronous ones. Furthermore, the standard DOM interface exposes many synchronous methods, e.g., to access persistent storage, register event handlers, and mutate the DOM tree. TreeHouse and PSJ expose virtualized DOM interfaces, but these interfaces can only export asynchronous methods. Thus, porting DOM-intensive legacy applications to TreeHouse and PSJ is challenging, and writing new applications for these frameworks requires programmers to learn a new set of DOM semantics. In contrast, Pivot's synchronous interfaces make it easy to port old applications and write new ones that use familiar browser APIs. For example, the Pivot application from Section V-B accesses DOM storage using the standard synchronous interface.

PostMash [4] is another framework which uses frames as isolation containers and `postMessage()` as the cross-domain communication primitive. To include an untrusted library, an integrator page includes a JavaScript stub written by the library's developer; this stub exports the library's API, and implements RPC requests and responses via `postMessage()`. Since the stub runs in the integrator's frame, it must be trusted. In contrast, Pivot places all code from untrusted libraries in external frames, meaning that the integrating master frame does not need to trust any code from the satellites. Pivot also provides synchronous interfaces, unlike PostMash. However, PostMash does not rewrite any code, so it avoids the overhead of the dynamic checks that Pivot inserts at function call sites.

SMash [15] uses frames as isolation containers, but it leverages fragment identifiers in URLs as the cross-frame communication primitive. This communication channel

leverages the fact that, in older browsers, parent frames and child frames could modify each other's URLs, even if they were from different origins. This technique was popular in older browsers, before `postMessage()` was standardized. However, fragment communication no longer works on many popular browsers, and `postMessage()` is now the standard mechanism for cross-frame communication.

Rewriting-based isolation: Caja [24] and Jigsaw [22] use code rewriting to safely place mutually untrusting code within the same frame. Like Pivot, both systems provide synchronous RPCs. However, Section V shows that Pivot can provide superior performance by *selectively rewriting code*—only the synchronous application core needs to be rewritten, and untrusted libraries that do not make synchronous RPCs can be left unrewritten (and thus free of slow, dynamic security checks).

Pivot only supports pass-by-value cross-domain sharing. In contrast, Caja and Jigsaw provide a brokered form of pass-by-reference in which a wrapper object mediates untrusted accesses to a sensitive underlying object.

Converting asynchronous JavaScript to synchronous JavaScript: Narrative JS [25] uses rewriting to transform asynchronous JavaScript code into a pseudo-synchronous form. However, Narrative JS requires programmers to use a special syntax for asynchronous calls; more generally, it requires developers to explicitly reason about how to safely execute interleaved threads within the same frame. This breaks the standard JavaScript programming model in which each frame has only one thread. In contrast, Pivot's true synchronous interfaces avoid the race conditions that are endemic to pseudo-synchronous frameworks [18], [22]. The Narrative JS compiler does not leverage the `yield` statement that is built into the JavaScript engine, so it has to manually generate JavaScript-level code to manage generator state, and only top-level functions in call chains can yield. Narrative JS is not a security framework, so it does not provide features like cross-domain isolation or RPC registration interfaces. Similar rewriting solutions include jwacs [36] and StratifiedJS [27].

The task.js library uses the built-in `yield` statement to provide a cooperatively multithreaded environment within a single frame [26]. However, only the top-most function in a call chain is allowed to yield; in contrast, Pivot uses rewriting to allow *any* function in a call chain to be an RPC. task.js requires programmers to explicitly insert yields and use a special syntax to manage execution threads. Pivot introduces no new keywords or concurrency semantics. task.js provides no security mechanisms to isolate untrusted code, nor does it provide performance isolation—since all tasks run in the same frame, a buggy or malicious task can launch a denial-of-service attack on other tasks by spin-looping.

Other systems: Pivot runs on today's commodity browsers, leveraging frames, `yield`, and `postMessage()` to provide synchronous, pass-by-value sharing between domains. The resulting security model is simple, and a natural fit for many applications. Other mashup systems provide more complicated (but more expressive) policy languages. For example, several frameworks modify the JavaScript engine to support transactional JavaScript, such that, if untrusted code violates a security policy, the effects of that code can be rolled back [8], [29]. Mash-IF [17] uses an IFC approach, associating each domain's JavaScript objects with security labels, and using a modified browser to track taint and enforce disclosure policies. OMash [7], object views [18], Embassies [13], and MashupOS [34] provide additional models for expressing cross-domain security policies.

VII. Conclusions

Pivot is a new isolation framework for web applications. Pivot uses iframes as isolation containers, but leverages rewriting to provide synchronous cross-domain RPCs. To provide synchronicity, Pivot translates a frame's JavaScript code into a generator function. These generators are explicitly invoked by Pivot's distributed cross-frame scheduler; this scheduler layers an RPC mechanism atop `postMessage()` calls. Each satellite frame exposes a public interface by registering local function references with Pivot. When a satellite issues an RPC, its generator yields to the Pivot scheduler, which then sends a `postMessage()` to the satellite that implements the RPC. When Pivot receives a response, it restarts the caller's generator function.

Using the built-in isolation mechanisms provided by iframes, Pivot can avoid rewriting libraries that respond to, but do not make, synchronous RPC calls. By only rewriting an application's small, trusted core, Pivot avoids the performance penalties incurred by rewrite-everything frameworks which have to modify all code to ensure safety. Experiments show that Pivot RPCs are up to two orders of magnitude faster than those of Jigsaw, a state-of-the-art rewrite-everything framework. Compared to other rewriting solutions that implement pseudo-synchronous RPCs, Pivot provides true synchrony without forcing developers to use special function call syntax, a restricted version of the JavaScript language, or new concurrency semantics.

References

[1] B. Adida, A. Barth, and C. Jackson. Rootkits for JavaScript Environments. In *Proceedings of the USENIX Workshop on Offensive Technologies*, Montreal, Canada, August 2009.

[2] A. Adya, J. Howell, M. Theimer, W. Bolosky, and J. Douceur. Cooperative Task Management without Manual Stack Management. In *Proceedings of USENIX ATC*, Monterey, CA, June 2002.

[3] D. Akhawe, P. Saxena, and D. Song. Privilege Separation in HTML5 Applications. In *Proceedings of USENIX Security*, Bellevue, WA, August 2012.

[4] A. Barth, C. Jackson, and W. Li. Attacks on JavaScript Mashup Communication. In *Proceedings of Web 2.0 Security and Privacy*, Oakland, CA, May 2009.

[5] P. Boere. DOM-Storage-Query-Language: A SQL-inspired interface for DOM Storage. http://code.google.com/p/dom-storage-query-language/, 2011.

[6] Compuware. Compuware AJAX Edition: Free Web Performance Analysis and Debugging Tool. http://www.compuware.com/en_us/application-performance-management/products/ajax-free-edition/overview.html, 2014.

[7] S. Crites, F. Hsu, and H. Chen. OMash: Enabling Secure Web Mashups via Object Abstractions. In *Proceedings of CCS*, Alexandria, VA, October 2008.

[8] M. Dhawan, C.-C. Shan, and V. Ganapathy. Enhancing JavaScript with Transactions. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, Beijing, China, June 2012.

[9] Ecma International. Draft ECMAScript 6 Specification: Generators. http://wiki.ecmascript.org/doku.php?id=harmony:generators, February 20, 2014.

[10] A. Gavrilov. Bubblemark animation test: Silverlight (JavaScript and CLR) vs DHTML vs Flash (Flex) vs WPF vs Apollo vs Java (Swing). http://bubblemark.com/, 2009.

[11] G. Gherardi. JsonRPCjs. https://github.com/gimmi/jsonrpcjs, 2012.

[12] Google. Google-Caja: Performance of cajoled code. http://code.google.com/p/google-caja/wiki/Performance, October 4, 2011.

[13] J. Howell, B. Parno, and J. Douceur. Embassies: Radically Refactoring the Web. In *Proceedings of NSDI*, Lombard, IL, April 2013.

[14] L. Ingram and M. Walfish. TreeHouse: JavaScript Sandboxes to Help Web Developers Help Themselves. In *Proceedings of USENIX ATC*, Boston, MA, June 2012.

[15] F. D. Keukelaere, S. Bhola, M. Steiner, S. Chari, and S. Yoshihama. SMash: Secure Component Model for Cross-Domain Mashups on Unmodified Browsers. In *Proceedings of WWW*, Beijing, China, April 2008.

[16] O. Kiselyov, S. Peyton-Jones, and A. Sabry. Lazy v. Yield: Incremental, Linear Pretty-printing. In *Proceedings of APLAS*, Kyoto, Japan, December 2012.

[17] Z. Li, K. Zhang, and X. Wang. Mash-IF: Practical Information-Flow Control within Client-side Mashups. In *Proceedings of DSN*, Chicago, IL, June 2010.

[18] L. Meyerovich, A. Felt, and M. Miller. Object Views: Fine-grained Sharing in Browsers. In *Proceedings of WWW*, Raleigh, NC, April 2010. ACM.

[19] L. Meyerovich and B. Livshits. ConScript: Specifying and Enforcing Fine-grained Security Policies for JavaScript in the Browser. In *Proceedings of the IEEE Symposium on Security and Privacy*, Oakland, CA, May 2010.

[20] J. Mickens. Silo: Exploiting JavaScript and DOM Storage for Faster Page Loads. In *Proceedings of USENIX WebApps*, Boston, MA, June 2010.

[21] J. Mickens. Rivet: Browser-agnostic Remote Debugging for Web Applications. In *Proceedings of USENIX ATC*, Boston, MA, June 2012.

[22] J. Mickens and M. Finifter. Jigsaw: Efficient, Low-effort Mashup Isolation. In *Proceedings of USENIX WebApps*, Boston, MA, June 2012.

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

[24] M. Miller, M. Samuel, B. Laurie, I. Awad, and M. Stay. Caja: Safe active content in sanitized JavaScript. Google white paper. http://google-caja.googlecode.com/files/caja-spec-2008-06-07.pdf, June 7, 2008.

[25] N. Mix. Narrative JavaScript. http://www.neilmix.com/narrativejs, 2014.

[26] Mozilla. task.js: Beautiful Concurrency for JavaScript. https://github.com/mozilla/task.js, March 21, 2013.

[27] Oni Labs. StratifiedJS: JavaScript plus Structured Concurrency. http://onilabs.com/stratifiedjs, 2011.

[28] T. Parr. *The Definitive ANTLR Reference*. Pragmatic Bookshelf, Raleigh, North Carolina, 2007.

[29] G. Richards, C. Hammer, F. Nardelli, S. Jagannathan, and J. Vitek. Flexible Access Control for JavaScript. In *Proceedings of OOPSLA*, Indianapolis, IN, October 2013.

[30] E. Stark, M. Hamburg, and D. Boneh. Symmetric Cryptography in JavaScript. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, Honolulu, HI, December 2009.

[31] D. Synodinos. ECMAScript 5, Caja and Retrofitting Security: An Interview with Mark S. Miller. http://www.infoq.com/interviews/ecmascript-5-caja-retrofitting-security, February 25, 2011.

[32] R. von Behren, J. Condit, F. Zhou, G. Necula, and E. Brewer. Capriccio: Scalable Threads for Internet Services. In *Proceedings of SOSP*, Lake George, NY, October 2003.

[33] W3C Web Apps Working Group. Web Storage: W3C Working Draft. http://www.w3.org/TR/webstorage/, July 30 2013.

[34] H. Wang, X. Fan, J. Howell, and C. Jackson. Protection and Communication Abstractions for Web Browsers in MashupOS. In *Proceedings of SOSP*, Stevenson, WA, October 2007.

[35] Web Hypertext Application Technology Working Group (WHATWG). HTML Living Standard, Section 4.8.2: The iframe element. http://www.whatwg.org/specs/web-apps/current-work/#attr-iframe-sandbox, March 13, 2014.

[36] J. Wright. jwacs: Javascript With Advanced Continuation Support. http://chumsley.org/jwacs/, 2006.