Using Static Analysis for Ajax Intrusion Detection

Arjun Guha
Brown University
arjun@cs.brown.edu

Shriram Krishnamurthi
Brown University
sk@cs.brown.edu

Trevor Jim
AT&T Labs-Research
trevor@research.att.com

ABSTRACT

We present a static control-flow analysis for JavaScript programs running in a web browser. Our analysis tackles numerous challenges posed by modern web applications including asynchronous communication, frameworks, and dynamic code generation. We use our analysis to extract a model of expected client behavior as seen from the server, and build an intrusion-prevention proxy for the server: the proxy intercepts client requests and disables those that do not meet the expected behavior. We insert random asynchronous requests to foil mimicry attacks. Finally, we evaluate our technique against several real applications and show that it protects against an attack in a widely-used web application.

Categories and Subject Descriptors

D.2.4 [**Software/Program Verification**]: Reliability

General Terms

Languages Security

Keywords

JavaScript, Ajax, Control-Flow Analysis, Intrusion Detection

1. INTRODUCTION

Web applications that use client-side scripting are nearly ubiquitous: today, 98 of the 100 most-viewed web sites in the US use client-side JavaScript, and half of these use XMLHttpRequest (XHR), the asynchronous callback mechanism that characterizes *Ajax* web applications. The attraction of Ajax applications is that they can have a richer user interface and lower latency. However, they are also vulnerable to new kinds of attacks.

In an Ajax web application, the web server exposes a complex API to the client via a set of URLs. The client-side JavaScript and the server-side program are written together, and the server may expect the client to invoke the URLs in a particular sequence, with particular arguments. A malicious client, however, can invoke the URLs in any order and with any arguments. This paper presents a technique to mitigate such attacks.

WWW 2009, April 20–24, 2009, Madrid, Spain.
ACM 978-1-60558-487-4/09/04.

We have built a static control-flow analyzer for the client portion of Ajax web applications. The analyzer operates on the HTML and JavaScript code that constitute the program executed by the web browser, and it produces a flow graph of URLs that the client-side program can invoke on the server. We install this *request graph* in a reverse proxy that monitors all requests; any request that does not conform to this graph constitutes a potential attack. Our tool can prevent many common cross-site scripting (XSS) and cross-site request forgery (CSRF) attacks.

Our work is inspired by that of Wagner and Dean [33] and others (see §9), who used program analysis to build system call monitors and intrusion detectors. We share the following strengths and weaknesses with that line of research: we obtain a monitor automatically, without training against test data; we use conservative analyses to avoid false positives; we obtain significant coverage, but do not achieve complete coverage, e.g., we cannot stop a client from leaking data to third parties, and mimicry attacks [33] in particular require careful countermeasures (see §6 and §8).

Browser-based JavaScript poses many novel challenges for our analysis. For example, JavaScript programs interact with the browser via the Document Object Model (DOM), a representation of the web page shown to the user; the DOM is event-driven and accepts asynchronous inputs from the user. Different browsers can implement the DOM in very different ways, so developers use large JavaScript frameworks to mask these differences. Programs interact with the server via an asynchronous callback mechanism, XHR; sequential requests from the client can arrive out-of-order at the server. To reduce latency, clients often load or create scripts dynamically.

To the best of our knowledge, ours is the first effective program analysis for non-trivial JavaScript programs. Our analysis does not rely on knowledge of the server-side program, so it can be used with a variety of server technologies. The results of this program analysis are, of course, useful in other contexts such as optimization, debugging and documentation, which we do not discuss here.

To summarize our contributions: We analyze real-world, browser-based, event-driven JavaScript programs. We construct a graph that captures a well-behaved client's interactions. We construct a proxy to monitor the client. To defend against mimicry attacks, we insert random asynchronous requests while preserving behavior; we perform this obfuscation efficiently, and discuss how it also improves the monitor's performance. Finally, we show that these techniques scale to working applications and detect attacks on them. After describing our work, we discuss its limitations in §8.

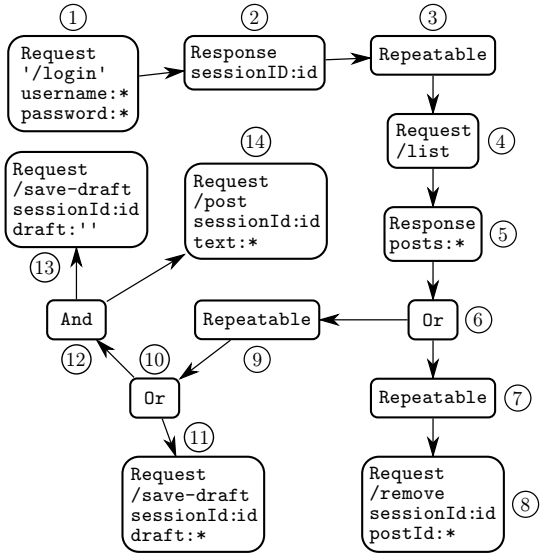

Figure 1: Request graph for a blog

2. MOTIVATING EXAMPLE

We use part of a blogging system written for a student homework assignment as a running example. The application supports multiple blogs and authentication: anyone can read posts, but only authors can create, edit, or delete posts. The application asynchronously saves drafts of posts.

Most of the data sent to the server is unstructured, user-generated content (the posts). However, the *requests* (in this paper, always from client to server) and *responses* (server-to-client) are themselves structured, and there is structure to the sequence of messages. This structure is modelled by the *request graph*, whose nodes specify the expected message structure and edges specify the sequence of messages.

Figure 1 shows the request graph that our tool generates for the application (with a few uninteresting nodes elided for simplicity). From the server's perspective, the client-side code executes according to the graph, starting at the root node 1, where it sends a login request to the server. When the login succeeds, the client receives a session ID response from the server (node 2), and then enters a loop (indicated by the **Repeatable** node 3) where it obtains a listing of the blog posts from the server, and then lets the user nondeterministically select from various activities (creating, deleting, editing posts). The nondeterministic choice is indicated by the **Or** node 6.

This simple graph illustrates several challenges that our analysis must overcome:

- When an author creates a new post, the client-side code first makes a request to publish the content, then makes a second request to clear the auto-save backup. However, the two requests can arrive at the server in either order, so we use the **And** node 12 to compose them. Requests may interleave in general; only dominating responses order them (§3.1).

- The server's request API is defined by a collection of URLs (`/login`, `/post`, `/remove`, etc.) and associated arguments. Our analysis must statically determine the structure of URLs and their arguments to correctly distinguish requests (§3.2).

- When an author successfully logs in, the blog returns a capability string that is checked by subsequent actions. This is the `sessionId` field found in several nodes. Our analysis tries to identify such capabilities (§3.5) so the monitor can track them (§5).

The remainder of the paper shows how we handle these challenges, as well as many others that we have encountered in real-world applications.

3. THE CONTROL-FLOW GRAPH

Our first step is to extract a control-flow graph from the HTML and JavaScript that make up the client side of the web application. (§4 shows how we generate request graphs from control-flow graphs.)

A control flow analysis for JavaScript must account for higher-order functions and, similarly, method dispatch. We use the *uniform k-CFA* algorithm [23], which is an established technique for higher-order control flow analyses.

CFA models a program's functions as abstract values. Expressions in a program are mapped to sets of abstract values. Collectively, these sets form an abstract heap. By explicitly modelling the heap, CFA sidesteps aliasing issues.

Statements in a program are interpreted as constraints that flow values between value sets in the abstract heap. By repeated application of these constraints, an abstract function can flow from its definition to its use, through arbitrary sequences of higher-order functions. Constraints are applied until the heap reaches a fixed-point.

3.1 DOM Events and Asynchronous Requests

Browsers execute JavaScript in an event-driven fashion. Events include asynchronous XHR callbacks and DOM events that are triggered by user interactions. As events occur the browser chooses one and executes a corresponding event handler; after the handler finishes, the browser chooses another available event, and so on.

Every handler is an entry point of the program, and handlers execute in some sequence. Our analysis cannot predict the exact sequence, because it depends on user interactions. However, partial information is available. For instance, figure 1 indicates that users must log in before they can post. We incorporate this by assuming that handlers enter the control-flow graph only when the handler is installed in the DOM. (Currently we do not model removing or disabling handlers, which introduces some imprecision.)

Figure 2 shows a typical Ajax event handler. It asynchronously sends a blog post to the server and tells the server to clear the current draft. It motivates several subtleties in our analysis:

- Every server request goes through XHR; a context-insensitive analysis would merge all of these flows. It would not be sufficient to use a special case for XHR, because XHR is often called indirectly by helper functions. Therefore, we use a context-sensitive analysis, k-CFA.

- The URL and the request data are specified at distinct method calls, `open` and `send`. Our analysis treats these specially, recording the URL and request data in the abstract request object, so they can be correlated.

```
var postHandler = function(event) {
  var postReq = new XMLHttpRequest();
  postReq.open('/post');
  postReq.onreadystatechange =
    function(responseText) {
      if (postReq.readyState == 4) {
        alert('Post successful.'); }};
  postReq.send(... post data ...);
  var clearReq = new XMLHttpRequest();
  clearReq.open('/draft');
  clearReq.onreadystatechange = function(_) {
    return; }; // nothing to do
  clearReq.send(''); } // save an empty draft

var postButton = document.getElementById('post');
postButton.addEventListener('click',postHandler);
```

Figure 2: Event handler for Post operations

- If we treat the server's response (`responseText`) as entirely indeterminate, the results of our analysis will be extremely imprecise. §3.2 describes how, instead, we can infer the structure of values sent by the server.

3.2 Structured Data

An XHR object sends and receives data as strings, which are usually serialized forms of structured data. In principle, a sophisticated analysis can recover some of this structure [30]. In practice, developers know what marshalling techniques their applications use. Therefore, we ask the users of our analysis to tell us how data is marshalled, and we specialize the analysis to obtain better precision. We have implemented specializations for URL encoding and, more interestingly, JavaScript Object Notation (JSON) [5].

For the JSON case, it is relatively easy to determine the structure of the strings sent to servers. The key observation is that the strings are obtained by a simple serialization of JavaScript objects. Therefore, we can obtain their structure by looking at the (abstract) object being serialized and ignoring serialization itself (by providing a stub §3.4). Though the values of the fields of these objects may contain indeterminate values (because they contain, for instance, user form inputs), the structure of the object itself is usually statically determinate. We thus increase the precision of our security monitor by matching actual requests seen at runtime against abstract requests determined during static analysis. Ignoring serialization libraries has an additional benefit: these libraries tend to be complex and employ reflection heavily, introducing considerable imprecision into the results of the analysis.

Responses are more difficult. A client typically unmarshalls a JSON string using `eval` or a complicated JSON parser:

```
req.onreadystatechange = function () {
  if (req.readyState == 4) {
    result = eval(req.responseBody); ...
```

Here, `req.responseBody` is the server's (indeterminate) response string, and so to our analysis, `result` is an indeterminate object. However, the desired shape of `result` is indirectly expressed by the application's field lookups. For example, if the application sends a login request, the response-processing code might contain this fragment:

```
if (result.success == true) {
  sessionId = result.sessionId; ...
```

This code expects an object bound to `result` that, if defined, contains a `success` field, whose truth implies the existence of a `sessionId` field, and so on. The constraints generated by our static analysis formally express this intuition: when an indeterminate object flows into an expression that accesses a field (here `result.success` and `result.sessionId`), the analysis adds the field to the indeterminate object, and populates it with a fresh indeterminate value.

This is effectively a very lightweight shape analysis [28]. JSON strings represent syntactic values containing only fields, not methods; the resulting objects can be treated like any other object. The soundness of this technique depends on assuming that the server is well-behaved and returns pure JSON only, as we discuss further in §8.

3.3 Dynamically-Generated JavaScript

A web page can modify itself using JavaScript, and the script can even add new *code* to the page, which the browser will load and execute on the fly. This has many uses: for example, it is used to decrease initial load times and to load browser-specific code.

Our analysis must handle dynamically-generated scripts, because they can make requests to the server; ignoring them would cause false positives—legitimate requests that our proxy will reject. We use three techniques that work well in practice.

First, we track string constants and precisely model basic string operations such as string concatenation. We apply a permissive HTML parser to identify dynamically loaded scripts in these strings. In our experience, simply tracking constants and concatenation is sufficient to detect most scripts. We don't precisely model the semantics of more complex string operations.

Second, we handle a common special case where the code is generated from some external script, e.g.,

```
"<script src='" + file + "'></script>"
```

Usually `file` is some script known to the developer residing on the server, but not known to our analysis. In this case, our analysis asks the developer for the file.

Third, there are some cases where our analysis is not able to completely determine the structure of the generated code, but the content of the "holes" cannot affect the soundness of the analysis. For example, here is a common case used in AjaxIM [14]:

```
"<span id=" + id
  + "onclick='handleClick(" + id + "')>"
```

The code constructs a new DOM node with an unknown, unique `id`, and installs a known handler (`handleClick`) for it. In this case, the actual value of the `id` cannot change the results of the analysis, so we can proceed without it.

These techniques do not completely solve the problem of dynamically-generated code, which is undecidable in general. We discuss this further in §8.

3.4 Frameworks

In practice, there is no such thing as *the* DOM; different browsers implement it differently, and web applications and our analysis must take this into account. Applications use JavaScript frameworks to cope with these differences. These frameworks check which browser they are running on, and

GraphNode ::= **Begin**
| **Request**(*Url**, *Value**)
| **Response**(*Value**)
| **And**
| **Or**
| **Jump**(*Url**)
| **Repeatable**

Figure 3: The grammar of request graph nodes

present a uniform API that hides browser differences. Consequently, frameworks use unusual coding tricks which make analysis results imprecise, justifying special treatment.

We model the DOM as implemented by Firefox, so when a framework checks for the browser version, our tool picks the Firefox branch. This means our analysis does not explore all branches, which would cause imprecision. Supporting more browsers is simply a matter of elbow grease.

Our analysis sometimes uses a "stub" framework in place of a real framework. The stub framework replaces some real framework functions with skeletal JavaScript that captures essential control flows. In some cases, we have hard-coded the analysis for some portions of frameworks. This is similar to the use of stubs in model checkers, where it helps control the state explosion problem.

We discuss our experience with specific frameworks in §7.

3.5 Capabilities

Upon authentication, many services return session identifiers and similar "capabilities." Capabilities are created by the server and are passed back to the server by the client. For instance, in §2, the blog provides and expects a session ID. Enriching the monitor to track capabilities improves protection against malicious requests.

In general it is difficult to know what is a capability and what isn't. This might require programmer annotation, which we try to avoid, or knowledge of server-side technology, to which we are agnostic.

Fortunately, there is a cheap and practical alternative. We adopt the heuristic that "capabilities" are just values that are received in a response and sent, unmodified, in subsequent requests. Given the analysis of requests and responses (§3.2), identifying these value-flows is simple and can be done *automatically* by looking for the above flow pattern. In figure 1, for instance, the *id* returned at node 2 is sent with subsequent requests (nodes 8, 11, 13, and 14). We describe how to monitor capabilities in §5.

4. THE REQUEST GRAPH

The analysis of §3 produces a control-flow graph of the client program's behavior. This control-flow graph must be transformed into a request graph like the one in figure 1.

Figure 3 gives a grammar for the nodes of the request graph. **Begin** represents the beginning of the program. **Request** nodes describe requests made to the server, while **Response** nodes describe expected server responses. The monitor expects to see all the children of an **And** node, but in indeterminate order. In contrast, an **Or** node corresponds to program conditionals, so the monitor expects to see only one branch taken (where no two branches are identical). **Jump** nodes represent transitions to other pages.

1. Insert **Request** nodes immediately after calls to `XMLHttpRequest.send`.
2. While there exists a vertex v that represents a JavaScript statement or expression,
   $\forall v_p, v_s$ s.t. the control-flow graph has edges (v_p, v) and (v, v_s), insert the edge (v_p, v_s) and delete the vertex v.
3. For each **Request** node r:
   (a) Create a new **And** node, a.
   (b) Substitute r with a. a therefore assumes the successors and predecessors of r, which is now disconnected from the rest of the graph.
   (c) Create the edge (a, r).
4. Add an edge from each **Request** node to its associated **Response** node.
5. Replace all nodes that represent DOM events with **Repeatable** nodes.

Figure 4: Transforming control-flow graphs into request graphs

Repeatable indicates that its descendant may repeat indefinitely; it is used to model the request handlers of DOM nodes. In principle, this cycle can be handled by the graph itself. We retain **Repeatable** nodes for two reasons. One is its use in monitoring, as we discuss in §5. The other is that **Repeatable** also handles loops that arise from user interaction with the DOM. Any remaining cycles in the request graph suggest that a (possibly mutually-)recursive function is making server requests. These seem to be rare in JavaScript programs—we have not encountered any in the applications in §7—and therefore serve as a diagnostic to the developer.

Generating the Request Graph. Figure 4 presents our algorithm to construct request graphs from control-flow graphs. The main idea is to focus on the sub-graph obtained by eliding all the operations that execute entirely on the client. This is Step 2 of our algorithm: we filter out all nodes that represent statements and expressions, which cannot be observed by the server. However, we must also apply several other transformations to the control-flow graph.

First, applications construct server requests in steps, by creating an XHR object, and then invoking its `open` and `send` methods. For each request we therefore create a **Request** node, which contains the URL and the request data, and insert it immediately after the `send` method is applied. This is Step 1 of our algorithm.

Second, the order in which the server sees requests can differ from the order in which the client sends them. When an application makes multiple asynchronous requests, our control-flow graph sequences them in the order in which they are sent. The server may, however, receive them out of order, and this must be reflected in our request graph. Step 3 of our algorithm inserts **And** nodes allow reordering of requests. We have to ensure that this transformation is done correctly in the presence of branching in an event handler.

Third, each **Response** handler may be invoked immediately after its associated **Request** handler. In Step 4, we make this dependency explicit by adding an edge from each **Request** node to its associated **Response** handler.

Finally, every DOM event handler is an entry point into the program that may be reentered arbitrarily based on user-interactions. In Step 5, we make this explicit by replacing all nodes that represent DOM events with **Repeatable** nodes.

5. THE SECURITY MONITOR

The monitor—which we implement as a reverse proxy—ensures that the sequence of requests that the server receives at runtime matches a sequence of abstract requests in the request graph. We first discuss handling the lack of determinism, then address the treatment of capabilities.

Nondeterministic Request Graphs. **Or** nodes in the request graph represent nondeterministic choice, while **And** nodes represent nondeterministic order. In addition, a given concrete request can match several **Request** nodes (due to over-approximation). Matching real requests to nodes in the request graph is thus similar to evaluating against an and-or tree, à la Prolog. There are, however, two important differences: because the input is a potentially infinite stream of requests, our matching is on-line (in the algorithmic sense), not against a fixed and known set of facts; and we are matching against a graph, not a tree. We omit the formal algorithm due to lack of space.

One salient detail is the treatment of **Repeatable**. Say a **Repeatable** dominates an **And** node with child nodes A and B. Suppose the first request matches B and not A, and the second matches both A and B. The monitor could then be in any of three states: it has seen both an A and a B in the first iteration and is now beginning its second iteration; it has seen a B in the first iteration and an A in the second iteration; or, it has seen a B in the first iteration and a B in the second iteration.

The monitor could allow all three interpretations, but the number of choices grows rapidly. We therefore assume, in the above example, that the second request matched the first iteration's A, and that the monitor is now beginning its second iteration. This interpretation is expedient but, in general, not equivalent to the fully-nondeterministic interpretation. We therefore make this an option that the developer can choose. In cases where this properly reflects the program's behavior—e.g., when the analysis was not able to detect that a response dominated the next set of requests—this interpretation loses no generality.

Monitoring Capabilities. The analysis identifies capabilities (§3.5) and marks them in the request graph (e.g., in figure 1). The monitor can then look for capabilities in server responses, record them, and verify that subsequent client requests supply matching capabilities. The monitor also needs to know when to discard a capability. In practice the only capabilities we have tracked are session IDs, which can be discarded when the session expires.

6. MIMICRY ATTACKS

The monitor enforces a client's normal pattern of requests, but cannot prevent attacks that work within this normal pattern—mimicry attacks. Many web applications have "console nodes" or "event loops" that constitute a good basis for a mimicry attack. Consider, for example, our blogging application (figure 1). We wish to prevent, say, a cross-site request forgery attack, where an attacker causes the client to `/post` or `/remove` by means of an injected script. A clever attacker could get this past our monitor by first requesting a listing of all posts, placing our monitor at node 5, from which all requests are possible.

We can make this more difficult for the attacker by sending clients slightly different applications at different times. In particular, we want each session to have a different request graph and the additional requests to contain distinctive data; in turn, each modified request graph can result from several different client programs (with correspondingly different control-flow graphs). Overcoming this defense would require the attacker to embed a sophisticated program analysis within the attack itself.

In addition, this has the fortunate side-effect of improving the monitor's performance. The inserted random requests serve to distinguish between different request graph paths. Their receipt therefore prunes the space of nondeterminism.

Our tool inserts *guard requests* to accompany existing *application requests*. Issuing guard requests synchronously with their associated application requests could double the roundtrip time to the security proxy, so we issue guard requests asynchronously; the monitor uses a short queue to wait for both guard and application requests.

Note that this technique critically depends on asynchrony. Had a CGI-style application tried to use such a technique, users would have found it unreasonable to repeatedly click through "guard pages". Our technique is thus well-matched with the Ajax architecture. (We discuss a related technique, the insertion of null system calls [11], in §9.)

Client Program Mutation. We decide where to insert random requests by examining the request graph produced by our analysis. The analysis maintains a mapping between the request graph and the client source code, so that we can efficiently generate the modified client code as necessary.

To guard a chosen **Request** node in the request graph, we insert an additional request at a program point that dominates the corresponding actual request, and is dominated by any preceding request. If there are multiple such points, we choose one at random.

As mentioned in §3.4, during our analysis we replace some portions of frameworks with stubs that give us more precise control-flow for non-framework (application) code. The stubs exist only during analysis and their **Request** nodes should not be guarded. The source mapping identifies nodes that come from stubs, so we can avoid this.

Prototype Hijacking. *Prototype hijacking* [24] is a way to modify the behavior of any JavaScript method, including XHR. An attacker could use it to alter XHR to record all requests, and thereby discover how we have inserted random asynchronous guard requests.

We do not know of a foolproof method to prevent hijacking; it may require changes to the JavaScript language. We have experimented with a defense in Firefox and Safari and have found it to be effective against simple attacks. It is based on the observation that major browsers parse and execute scripts in the `<head>` in order [18]. This means that the first execution of the first `<head>` script cannot be corrupted by other scripts, and can access the uncorrupted XHR prototype. Our defense modifies the first script to store away the XHR prototype, and install a timed handler that periodically verifies that XHR has not been altered. It also modifies

the clearInterval method that removes timers by adding a wrapper that ensures that our timer is not being removed.

Relating the Two Techniques. We have now described two intrusion detection techniques: monitoring the request graph (which tracks the order of messages) and inserting random asynchronous guard requests (which makes it more difficult for clients to masquerade). Though we have presented the latter as a means to strengthen the former, the two are actually distinct methods that can be used independently or combined. In particular, sometimes a program's request graph has insufficient structure to serve as an effective deterrent. In this case we can still use the randomization technique to obtain some protection against malice.

Since these techniques are independent, it's worth asking whether a context-sensitive analysis is necessary if we are only interested in inserting guard requests. It is. We want to insert guard requests selectively, with potentially different guards for different requests. Without context sensitivity, potentially all invocations of the `send` method of an XHR object will be conflated. We cannot insert the guard request at that one point, because it would affect all requests. Instead, for each request, we have start from that `send` and follow the control-flow graph backwards to a point in code that is not reachable from such a reverse flow from any other request. Without context-sensitivity, the reverse flows from `send` will point to all its callers, making it impossible to find a program point unique to a particular request.

7. EVALUATION

To evaluate the effectiveness of our tools and techniques, we have applied them in several contexts beyond unit tests. Each stresses a different aspect of our work:

- Small, student-written blog applications whose request graphs are fairly straightforward to verify by hand (§7.2.1).
- Continue and Resume, real applications for which we have access to the developers, so we can evaluate the quality of the request graph against the opinion of an authoritative source (§7.2.2).
- AjaxIM, a widely-used application written by a third-party, in which our application is able to protect against a discovered attack (§7.2.4).
- Machine-generated code produced by the Google Web Toolkit, to determine whether the analysis can cope with the vagaries of generated code (§7.2.5).
- Two kinds of libraries: Prototype is representative of many common Ajax frameworks (§7.2.3), while Flapjax is unusually control-centric and hence demands more special treatment (§7.2.6).

7.1 Summary

Graph Quality. Our analysis is able to successfully construct non-trivial request graphs. For the portion of AjaxIM we analyzed, the request graph has 35 nodes; for Continue, it has 106 nodes; for Resume, 81 nodes. All have non-trivial sequences of requests before getting to potentially malicious nodes (such as those that write data). We discuss AjaxIM below; in Resume and Continue, even after login, there are at least two intervening requests between each write. A non-trivial request graph is, of course, necessary for the server to be able to detect request ordering violations. (§3.1 argued the need for context-sensitivity in the abstract. Our evaluation justifies this: without it, these graphs would have had roughly two nodes with no useful structure.)

Protecting Against Vulnerabilities. These graphs are actually effective. They successfully detected our injected attacks in the student blogs, Continue, and Resume.

Besides injected attacks, in the process of conducting these experiments we discovered a true vulnerability in AjaxIM that allows an arbitrary user to acquire administrative privileges. This is because neither server nor client sanitizes messages, so a malicious client can send a message that contains JavaScript to an administrator; this code is executed on display (i.e., a persistent XSS attack). In our experiments, we revoked the administrator's privileges and gave the attacker administrative privileges.

Without the security monitor, an attacker simply has to issue a request to toggle administrative privileges. The application structure, however, requires the administrator to first issue a search request to retrieve a list of users before administrative actions can be invoked. This dependency is captured in our request graph. Therefore, our monitor successfully protects against a basic XSS or CSRF attack.

In principle, a clever attacker could mimic the application's workflow. Even such an attacker, however, would face a significantly greater barrier due to our use of random requests (§6): this requires the attacker to determine the guard requests corresponding to both searching and setting permissions. We conjecture this requires the attack to contain program differencing [16], a technique that is difficult to implement effectively for modern languages.

Run-Time Overhead. The run-time overhead—introduced by the proxy—is minimal; even our prototype, unoptimized proxy induces a lag of less than half a second, and this can easily be reduced.

Analysis Time. The running-time of the analysis is much greater. The analysis of the GWT applications took between 28 seconds and 4 minutes (CPU time) on an Intel Core 2 Duo at 2 GHz. The administrator portion of AjaxIM took 45 minutes. The analysis of Continue and Resume took 2 minutes each. It helps that Continue and Resume are multi-page applications, so the analyzer effectively re-starts on each page and "pastes" together the results. This greatly improves tractability. We note that we have focused on handling a large set of features (§3) instead of engineering for performance. In addition, the use of a context-sensitive analysis is bound to be expensive, but we chose it (a) out of necessity (§3.1), and (b) because the analysis needs to run only once per code-release, not on every connection.

7.2 Details

7.2.1 Student-Written Blogs

The blog introduced in §2 is one of many written by students for an assignment in programming languages course. Although they are relatively simple, they exercise various JavaScript techniques, and reflect the diversity of being writ-

ten independently by a group of students with different backgrounds in Web programming.

We attacked the blogs' servers using both XSS and CSRF attacks. Since these are not production applications, this was easily done. However, once we applied our request graph tracing, this became substantially harder to do. Notably, randomization made a normal CSRF attack impossible, as a single CSRF cannot send both the application's request and the guard request.

7.2.2 Continue and Resume

We applied this work to two applications: Continue, a computer-science conference paper manager (`continue2.cs.brown.edu`), and Resume, a faculty-hiring job-search manager. Both run mostly on the client and provide features such as auto-saving of reviews. They are both in active use by actual, third-party users. Both were developed by a student at Brown University. The client portions of Continue and Resume are, respectively, 4.3 kloc and 2.9 kloc, of which 900 lines are in common libraries.

Both applications make extensive use of the Flapjax framework, which we discuss separately in §7.2.6. Though developed locally, they predate this project, so it is unlikely that we significantly biased their construction. They also have several real-world users who constantly demand new features, further reducing bias.

Because these are both applications for which we have extensive contact with the lead developer, we were able to perform a significant qualitative evaluation. We asked the lead developer to generate the request graphs for Continue and Resume. This confirmed our conjecture that developers grossly underestimate the degree of nondeterminism in the graph. We then discussed the generated graph and confirmed its accuracy in the opinion of the developer. Furthermore, we extensively tested the actual programs against a monitor using these graphs; normal operation triggered no violations, while attacks did.

7.2.3 The Prototype Framework

Prototype [25] is one of the most popular JavaScript libraries in use: in a 2007 Ajaxian survey, 34% of respondents indicated that they used Prototype. It is therefore imperative that our analysis handle it. It is also interesting because it does the bulk of its work by extending standard prototypes with useful functions, so it truly exercises our analysis (and, indeed, inspired the need for much of the rest of §3 as well).

To obtain a meaningful control-flow graph, we had to customize our analysis slightly. Prototype has two functions that are difficult to analyze: `$A` and `$w`. `$A` flattens arbitrary collections into arrays, using reflection and loops. `$w` uses regular expressions to break a string of words into an array of words. We would normally tolerate the default loose approximation of this function, but Prototype frequently uses it to transform a string into an array of property names to which it assigns values (!):

```
$w('fade appear grow ...').
  each(function(effect) {
    Effect.Methods[effect] =
      function(element, options){...}});
```

(By default `effect` would be indeterminate, thus adding a function to *all* properties of `Effect.Methods`.) Our analysis-specific versions of `$A` and `$w` total 31 loc.

We had to change a total of 200 loc out of 4 kloc. This included commenting out the definitions of `$A` and `$w`; transforming two general `for` loops into `for each` loops that the analysis can unroll; and hard-coding some browser dependency checks. Other than the changes to the two functions, the remainder could have been automated. After these changes our analysis handles Prototype effectively, as demonstrated by our analysis of AjaxIM (§7.2.4).

Having modified the library, we must address soundness. Testing for the browser is just partial evaluation. The two replaced functions, `$A` and `$w`, are simple, self-contained functions that are easy to rewrite as primitives. Therefore, we are confident that these changes do not hurt soundness.

7.2.4 AjaxIM

AjaxIM [14] is a browser-based instant-messaging service. The client communicates with its server using JSON and URL-encoded messages. AjaxIM challenges our analysis tools in two principal ways:

- The client uses many general-purpose frameworks that tend to be harder to analyze than application code. The frameworks are Prototype, script.aculo.ous, Prototype Window Class, SoundManager 2, and a JavaScript MD5 Library. These therefore significantly tested our handling of large libraries.

- The code makes extensive use of meta-programming. The only parts of its interface that are statically specified are dialog boxes such as login and registration screens. The interesting portions of the interface—chat windows and administration windows—are constructed in JavaScript by string concatenation and displayed on-demand. This forced our analysis to support dynamic loading, mutable environments, and other techniques that are commonly used in real applications.

To enable AjaxIM to pass our analysis, we had to perform one small modification to the MD5 library. We re-defined the MD5 function to immediately return an indeterminate string. Without this re-definition, the analysis would attempt an intractable exploration of call sequences in that library. All frameworks other than MD5 and Prototype where handled as-is.

In all, AjaxIM is one of the larger JavaScript applications. Besides Prototype, the other libraries are about 3 kloc. The core AjaxIM application is itself another 3 kloc of mostly JavaScript and HTML, resulting in an application of about 10k lines. Our analysis currently handles the administrative portion and can tackle related modules, but does not yet scale to the entire suite as a whole program.

7.2.5 Google Web Toolkit

Programming directly against the DOM is difficult and non-portable, so developers tend not to do so. We have applied our analysis to applications that use JavaScript frameworks to alleviate this problem. An alternative is to treat JavaScript as the target language for a compiler, such as the Google Web Toolkit (GWT) [13].

Program analyses often have implicit assumptions about the structure of programs that are biased by expectations of what *humans* normally write. Thus, they tend to be brittle in the face of machine-generated code. The situation is considerably worse with the GWT, because the generated code

dynamically loads code dependent on the browser in use. Since our analysis models dynamic loading (§3.3) and masquerades as a particular browser (§3.4), these issues don't adversely affect us. We were thus able to successfully apply our tool to the code generated from the sample applications included with GWT 1.4.61.

7.2.6 Flapjax

The Flapjax [31] library endows JavaScript with a reactive programming feel. In the Flapjax programming model, programmers do not write event-handling callbacks. Instead, they write functional expressions that refer to event sources (such as fields and buttons); the library converts these dependencies into a dataflow graph and automatically propagates changes through it, akin to a spreadsheet.

The Flapjax library, and programs that use it, make heavy use of the functional programming features of JavaScript. Programmers extensively employ higher-order functions and operators such as maps, reductions, and filters. Therefore, our ability to successfully handle clients of the Flapjax library is an indication of the strength of our modeling of higher-order control-flows.

The Flapjax library itself, however, taxes our analysis heavily in a peculiar way. Whereas most other libraries are concerned with data structures or DOM manipulation, the raison d'être of Flapjax is control-flow (namely, the creation and maintenance of the dataflow graph). As a result, the library results in a large number of control-flow nodes and edges that ultimately have no impact on the request graph.

To accelerate the analysis, we manually simplified Flapjax. Our analysis cannot track the fine-grained flows of computation through the priority queue that implements dataflow evaluation, resulting in excessive imprecision in the analysis output. It was relatively easy to remove this priority queue, turning a 5 kloc library into a 500-line stub. This made the analysis of Continue and Resume (§7.2.2) straightforward, while retaining sufficient precision.

Having modified the library, we must again address soundness. We were fortunate to be authorities on the implementation and design of Flapjax; this does mean we might not find it as easy to modify other libraries with complex control-flow, but then relatively few of these exist. More importantly, we carefully hand-constructed control-flow graphs for small programs, computed the corresponding request graphs, and confirmed that they match what the tool produces. Most significantly, the stubs resulted in monitors that worked successfully (§7.2.1, §7.2.2).

8. LIMITATIONS

Soundness. We would like to formally prove that our analysis is sound. A sound analysis would guarantee that our tool will never raise a false alarm, an important usability concern. However, a proof of soundness would require a formal semantics for JavaScript and the DOM in browsers, and this does not exist. It would be interesting to define this semantics, but that is a work unto itself, one that would have to take into account the many variations in browsers currently in the field.

We nevertheless claim that our approach is principled. First, we begin with well-known analyses that have been proven sound in their original context. Second, in applying these analyses we have listed the assumptions required for soundness, and have presented (informal) arguments for why the assumptions hold in our application. Third, whenever our monitor raises a false alarm, we immediately know that our analysis must be unsound, and can appropriately debug it. (Indeed, we found some such surprises—often based on nuances of the browser's DOM—in the process of testing our tool.) We are thus applying the experimental method, which is necessary when studying programs in the wild, as opposed to constructing a semantics in a vacuum.

There remain two practical concerns. The first is the use of dynamic loading and `eval`. Many uses of `eval` in applications are for parsing JSON strings, which we do model (§3.2). Similarly, we detect and manage dynamic code loading (§3.3). Like all other static analyses, however, we cannot trace arbitrarily complex string expressions that result in loading or evaluating code; our analysis bears this caveat, and should be complemented by dynamic methods. The second concern is the use of stubs. We present empirical evidence that our stubs do not cause unsoundness (§7.2.3 and §7.2.6). Since our stubs are written in JavaScript, relating them to the original framework code reduces to reasoning about program approximation, which is well studied.

Other Concerns. We have argued that attacking our techniques would require embedding an extremely sophisticated program analysis. There is one other avenue: to literally mimic the user's interactions (by simulating "pressing buttons", etc.). In the simplest of such attacks, the user would see the interface responding to actions they did not perform and should realize that something is awry. A more sophisticated attack could be carried out in a hidden frame.

Since our program analysis targets JavaScript exclusively, our evaluation is limited to applications that serve static HTML files and use Ajax for all dynamic content. However, our analysis of client-side JavaScript could be combined with analyses of various server-side scripting technologies.

Our monitor cannot distinguish between sessions automatically; instead, the user must provide an application-specific predicate that can distinguish sessions (which usually works by pattern-matching against the URL or examining cookies). It might be possible to infer this predicate from the request graph itself, and in special cases such as when using the GWT.

Though we do handle multi-page applications (§7.1), our request graph does not account for the use of browser operations such as the Back button. Prior work on verifying server-based applications that shows how to extend a control-flow model to accommodate browser operations [21] is compatible with the techniques in this paper.

As we discussed in §3.1, we do not model actions such as disabling an event-handler by disabling the associated DOM element. Because programs do disable events, recognizing this would lead to more precise monitors.

Our analysis currently handles JavaScript both in standalone files and inside HTML files (both in `<script>` tags and inlined in the HTML). It does not, however, handle other sources of JavaScript such as cascading style-sheets.

We handle data formats based on URL encoding and JSON, but not XML. This is mostly due to expediency: it saved the need for an analysis to handle XML data structures (which are superficially similar to JSON, which is also semi-structured, but introduce many more complexities such as namespaces). We would like to add such an analysis in future work.

9. RELATED WORK

System call monitoring is an established technique for intrusion detection in operating systems; our technique is similar, except it deals with remote requests instead. Some system call monitors use training on dynamic examples to obtain the model of calls to enforce [9]. Training has advantages but, as Gates and Taylor [10] indicate, also faces several problems, such as false-positives and the need for good test inputs (that reflect system changes over time).

Other systems use a model of calls constructed through static analysis [33, 7, 12, 26]. These systems all address operating system monitoring, where the operating system (corresponding to our server), monitor, and application (corresponding to our browser client) all run on the same machine. Therefore, there is no problem with requests being reordered during transmission. As a result, these approaches do not handle indeterminacy in the order of reception, as we must (§4). Also, these works do not discuss the use of random requests to obfuscate client program call sequences.

Giffen et. al [11] remotely monitor untrusted, distributed clusters. They do not need to address indeterminacy because their clients are entirely synchronous, so a new message is sent only after receipt of the previous one's response. Multithreading on the client would cause problems, but this work only monitors single-threaded applications; the paper says, "thread swaps will most likely cause the run-time model to fail." To curb nondeterminism, their work employs two techniques. The first is to insert *null system calls*. As we note, our random requests have the same effect (§6). In addition, they apply binary rewriting to assign unique names to call sites, to further curb nondeterminism in the monitor (an analogous technique, call-signing, is presented by Rajagopalan et. al [26]). Because invocations of XHR are, however, often deep within frameworks, signatures cannot be placed at invocations of that function; they must instead be "pushed up" the call chain to where the requests are really made. This is likely to require considerable precision from our control-flow analysis.

Giffin et. al [12] present *Dyck models* as an efficient technique for static analysis and suggest their use for remote intrusion detection. Applied naively, the Dyck model introduces sequencing, breaking asynchrony. A successful application of the Dyck model to Ajax applications might involve a cascade of asynchronous calls. However, such an attempt would require sophisticated program transformations to preserve semantics, and the authors do not discuss this.

Intrusion detection systems for web servers (e.g., [3]) work with HTTP and lower-level abstractions, sometimes incorporating models of server mechanics such as database and filesystem accesses. Our work models application-level semantics and is thus complementary.

Sharif et. al. [29] prove that intrusion detection systems that monitor control flow are strictly more precise than system call monitors. They discuss a technique for monitoring control flow events with an external monitor. Our system only monitors clients' requests, not control flow events. However, it may be possible to adapt their technique to monitor control flow events in the browser.

AjaxScope [20], BrowserShield [27], and CoreScript [35] secure browsers from known vulnerabilities by rewriting HTML and JavaScript. The JavaScript rewriting we use for inserting random requests (§6) is a simpler transformation and sufficient for our purposes, but we believe our technique for avoiding prototype hijacking attacks would be more robust if implemented as BrowserShield rewriting rules.

Static analyses have been applied to detect vulnerabilities in server-side web applications [1, 6, 19, 22, 34]. These complement our client-centric approach. These efforts handle only the procedural subset of PHP (Balzarotti et al. [1] manually transform the objects they encounter). In contrast, JavaScript is inherently object-oriented and higher-order, which makes objects unavoidable and analysis significantly harder. Jhala and Majumdar [17] provide a detailed model of interprocedural analysis of asynchronous programs, but for first-order programs. In contrast, Self [32] is another dynamically-typed, prototype-based language, but its most sophisticated analysis [2] relies heavily on dynamic feedback.

Swift/Jif [4] gives developers guarantees about information flow security. It assumes applications are written in Java augmented with information flow checks, and uses the GWT to compile these to JavaScript. This work is essentially incomparable to ours, since we do not place such a strong burden on developers of the server application.

10. FUTURE WORK

A great deal of client-side JavaScript code is concerned with presentation details, usually expressed through DOM manipulation. Because most DOM updates do not affect the request graph, we can model them as no-ops; but we still pay the price of analysis. If we could safely determine, using syntactic criteria, when code is uninteresting for control-flow analysis, we could skip a large volume of it. This would give us the scalability we will need as client-side JavaScript programs grow in size and complexity.

Our proxy currently maintains per-session state. This has two problems. First, we must be able to identify when a session begins and ends; we currently do this manually, and would prefer to obtain directives from the server application. Second, this use of state potentially hurts scalability. We would like to adopt techniques analogous to call-signing [11, 26] to reduce the use of state, but these techniques need to be generalized to handle higher-order languages.

We employ the Prefuse Toolkit [15] to visualize the control-flow and request graphs. While the visualization of the control-flow graph is most useful to us, it is the request graph that is most valuable to the developer. The current interface does not, however, provide feedback on, for instance, regions of the program where the degree of precision in the analysis should be increased or decreased.

The request graph is useful as a specification of potential client behavior. For instance, we can use it to create test inputs for the server. In particular, given that developers are known to be weak at reasoning about interleaving and ordering in concurrent contexts, the **And** nodes show different orderings that the server must be able to handle, some of which the developer may have overlooked.

As our performance shows (§7.1), analyzing JavaScript with contextual information can be quite expensive. Though JavaScript has no formal notion of modules, even informal modularity boundaries (such as web pages) help enormously. It should, therefore, be possible to employ modular flow-analyses that have been effective for other dynamically-typed languages [8] in this context.

11. ACKNOWLEDGEMENTS

We thank Spiridon Eliopoulos and Brendan Hickey, who helped implement the program analysis. David Brumley, Mihai Christodorescu, Larry Koved, Michael Steiner, Dan Wallach, and the anonymous reviewers provided useful feedback. This work was partially supported by Cisco, Google, and NSF grants CNS-0830945, CNS-0627310, and CCF-0447509.

12. REFERENCES

[1] D. Balzarotti, M. Cova, V. V. Felmetsger, and G. Vigna. Multi-module vulnerability analysis of web-based applications. In *ACM CCS Conference on Computer and Communications Security*, 2007.

[2] C. Chambers and D. Ungar. Iterative type analysis and extended message splitting: Optimizing dynamically-typed object-oriented programs. *LISP and Symbolic Computation*, 4(3), 1991.

[3] S. N. Chari and P.-C. Cheng. BlueBoX: A policy-driven, host-based intrusion detection system. *ACM Transactions in Information and System Security*, 6(2), 2003.

[4] S. Chong, J. Liu, A. C. Myers, X. Qi, K. Vikram, L. Zheng, and X. Zheng. Secure web applications via automatic partitioning. In *ACM SOSP Symposium on Operating Systems Principles*, 2007.

[5] D. Crockford. JSON. `http://www.json.org/`.

[6] M. Egele, M. Szydlowski, E. Kirda1, and C. Kruegel. Using static program analysis to aid intrusion detection. In *Detection of Intrusions and Malware and Vulnerability Assessment*, 2006.

[7] H. H. Feng, O. M. Kolesnikov, P. Fogla, W. Lee, and W. Gong. Anomaly detection using call stack information. In *IEEE Symposium on Security and Privacy*, 2003.

[8] C. Flanagan and M. Felleisen. Componential set-based analysis. *ACM Transactions on Programming Languages and Systems*, 21(2):369–415, 1999.

[9] S. Forrest, S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff. A sense of self for UNIX processes. In *IEEE Symposium on Security and Privacy*, 1996.

[10] C. Gates and C. Taylor. Challenging the anomaly detection paradigm: A provocative discussion. In *ACM New Security Paradigms Workshop*, 2006.

[11] J. T. Giffin, S. Jha, and B. Miller. Detecting manipulated remove call streams. In *USENIX Security Symposium*, 2002.

[12] J. T. Giffin, S. Jha, and B. P. Miller. Efficient context-sensitive intrusion detection. In *Network and Distributed System Security Symposium*, 2004.

[13] Google. GWT: Google web toolkit. `http://code.google.com/webtoolkit/`.

[14] J. Gross. Ajax IM. `http://www.ajaxim.com/`.

[15] J. Heer, S. K. Card, and J. A. Landay. Prefuse: A toolkit for interactive information visualization. In *ACM-SIGCHI Conference on Human Factors*, 2005.

[16] S. Horwitz. Identifying the semantic and textual differences between two versions of a program. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 234–245, 1990.

[17] R. Jhala and R. Majumdar. Interprocedural analysis of asynchronous programs. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2007.

[18] T. Jim, N. Swamy, and M. Hicks. Defeating script injection attacks with browser-enforced embedded policies. In *WWW International Conference on World Wide Web*, 2007.

[19] N. Jovanovic, C. Kruegel, and E. Kirda. Pixy: A static analysis tool for detecting web application vulnerabilities. In *IEEE Symposium on Security and Privacy*, 2006.

[20] E. Kıcıman and B. Livshits. AjaxScope: A platform for remotely monitoring the client-side behavior of web 2.0 applications. In *ACM SOSP Symposium on Operating Systems Principles*, 2007.

[21] D. R. Licata and S. Krishnamurthi. Verifying interactive Web programs. In *IEEE International Symposium on Automated Software Engineering*, pages 164–173, Sept. 2004.

[22] G. A. D. Lucca, A. R. Fasolino, M. Mastroianni, and P. Tramontana. Identifying cross site scripting vulnerabilities in web applications. In *IEEE Workshop on Web Site Evolution*, 2004.

[23] F. Nielson and H. R. Nielson. Infinitary control flow analysis: a collecting semantics for closure analysis. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1997.

[24] S. D. Paula and G. Fedon. Subverting AJAX. In *CCC Chaos Communications Conference*, 2006.

[25] Prototype Core Team. Prototype JavaScript Framework. `http://www.prototypejs.org/`.

[26] M. Rajagopalan, M. A. Hiltunen, T. Jim, and R. D. Schlichting. System call monitoring using authenticated system calls. In *IEEE Transactions on Dependable and Secure Computing*, 2006.

[27] C. Reis, J. Dunagan, H. J. Wang, O. Dubrovsky, and S. Esmeir. BrowserShield: Vulnerability-driven filtering of dynamic HTML. In *USENIX OSDI Symposium on Operating Systems Design and Implementation*, 2006.

[28] J. C. Reynolds. Automatic computation of data set definitions. In *Information Processing*, 1968.

[29] M. Sharif, K. Singh, J. Giffin, and W. Lee. Understanding precision in host based intrusion detection. In *RAID Recent Advances in Intrusion Detection*, 2007.

[30] Z. Su and G. Wassermann. The essence of command injection attacks in web applications. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2006.

[31] The Flapjax Team. Flapjax. `http://www.flapjax-lang.org/`.

[32] D. Ungar and R. B. Smith. Self: The power of simplicity. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 1987.

[33] D. Wagner and D. Dean. Intrusion detection via static analysis. In *IEEE Symposium on Security and Privacy*, 2001.

[34] Y. Xie and A. Aiken. Static detection of security vulnerabilities in scripting languages. In *USENIX Security Symposium*, 2006.

[35] D. Yu, A. Chander, N. Islam, and I. Serikov. JavaScript instrumentation for browser security. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2007.