# A Comprehensive Client-side Behavior Model for Diagnosing Attacks in Ajax Applications

Xinshu Dong[†], Kailas Patil[†], Jian Mao[‡*], and Zhenkai Liang[†]
[†]*School of Computing, National University of Singapore*
{*xdong, patilkr, liangzk*}*@comp.nus.edu.sg*
[‡]*School of Electronic and Information Engineering, BeiHang University*
*maojian@buaa.edu.cn*

***Abstract*—Behavior models of applications are widely used for diagnosing security incidents in complex web-based systems. However, Ajax techniques that enable better web experiences also make it fairly challenging to model Ajax application behaviors in the complex browser environment. In Ajax applications, server-side states are no longer synchronous with the views to end users at the client side. Therefore, to model the behaviors of Ajax applications, it is indispensable to incorporate client-side application states into the behavior models, as being explored by prior work. Unfortunately, how to leverage behavior models to perform security diagnosis in Ajax applications has yet been thoroughly examined. Existing models extracted from Ajax application behaviors are insufficient in a security context. In this paper, we propose a new behavior model for diagnosing attacks in Ajax applications, which abstracts both client-side state transitions as well as their communications to external servers. Our model articulates different states with the browser events or user actions that trigger state transitions. With a prototype implementation, we demonstrate that the proposed model is effective in attack diagnosis for real-world Ajax applications.**

## I. Introduction

Ajax is a technology that enables better performance and user experience for web applications. In Ajax web applications (or *Ajax applications* for short), rich functionality and prompt responsiveness to user actions are priorities in their design and implementation. However, such smoother experience comes from an evolution in the technology and paradigm in web application programming, resulting in a rather complex browser environment that supports the functionalities required by Ajax applications. In fact, the asynchronous nature of Ajax applications and the great complexity of modern browsers make it a daunting task to diagnose behaviors of Ajax applications under various attacks.

With traditional web applications, such as static or CGI-based web sites, it is straightforward to model their behaviors with HTTP requests sent to web servers. In those applications, each HTTP request typically represents a transition to a different server-side state. For example, the server of a webmail application may expect one request from a static HTML link to open the *inbox*, and a separate request from another link to open emails in the *sent* folder. Besides, these HTTP requests are only triggered when the corresponding links are clicked by the user. Therefore, behaviors of those applications can be simply modeled using the requests sent to web servers. Unfortunately, the situation is completely changed in Ajax applications, where requests may no longer explicitly change the state of the web application. Instead, HTTP requests in Ajax applications are largely *asynchronous*, sending information to web servers and fetching new data to the client side in the background. As a separate thread of execution, the client-side code uses the asynchronously fetched data to update the application view presented to the user. Traditional request-based behaviors models [1]–[3] do not apply to Ajax applications, since requests reveal little clue on the state transitions of Ajax applications.

On the other hand, it seems plausible to incorporate client-side information in modeling Ajax application behaviors. For example, although an Ajax-version of webmail application may not use separate requests to open the *inbox* and *sent* folder, the client-side code must switch the views between the two folders when the user clicks the corresponding tabs in the application. Based on such observations, prior research has explored constructing new models for Ajax applications by monitoring changes in the Document Object Model triggered by user clicks [4]. Such a model is helpful in crawling Ajax application pages and testing Ajax applications for various requirements [5]–[7].

However, such a model is constructed in a temporal order when user clicks trigger state transitions, with a focus on state exploration and reachability. It does not provide the necessary analysis capacity for runtime behavior diagnosis. As an example, when a cross-site scripting attack occurs in an Ajax application due to the inclusion of a third-party script, we need to trace its execution and identify the culprit script that initiates the attack. Therefore, such attack diagnosis requires precisely modeling the dependency between asynchronous events during the execution of Ajax applications. With such information, we can identify the *root cause* and *developments* of attacks in the application. The

*Research done when visiting National University of Singapore.

real challenge here is to extract necessary information to build up the interdependency between events occurring during execution of Ajax applications with the overwhelming complexity of the browser.

In this paper, we propose a new approach to diagnose attacks in Ajax applications by extracting *contextual* states of Ajax application behaviors from the browser environment. Despite the complexity of the browser runtime environment, our model accurately and concisely captures the *contexts* from where actions take place and the *transitions* between contexts. By maintaining the internal correlation between *user interactions*, *browser internal events* and *application actions*, our approach identifies the precise program context where any event or action is triggered, even in the asynchronous runtime environment of Ajax web applications. The constructed model can then be used to diagnose security incidents in Ajax applications, providing sufficient details from the entry of untrusted code, to the final attack action, such as sending a malicious request. To verify the applicability of our approach, we prototyped our solution in the Firefox web browser, and evaluated its effectiveness with several real-world web applications.

In summary, we make the following contributions in this work.

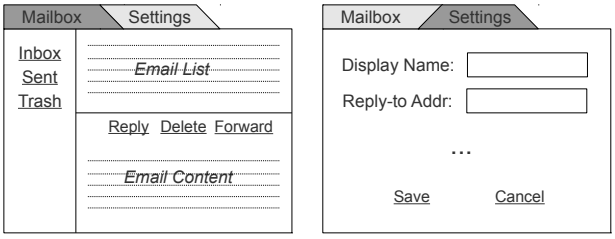
- We propose a new model for diagnosing attacks in Ajax applications with client-side modeling of the application behaviors.
- To extract the proposed models for Ajax applications, we develop a new approach that captures the detailed and precise correlation and dependency among user interactions, browser internal events, and application actions during the execution of Ajax applications.
- We build a prototype of our approach, and successfully apply it on behavior diagnosis of several real-world web applications with low performance overhead.

## II. Background and Motivating Example

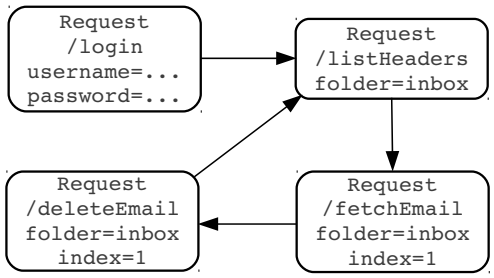
There are various prevailing threats to web application security, and we mainly target diagnosing malicious JavaScript that compromises the confidentiality and integrity of Ajax applications in this paper. Next, we will briefly introduce the security threats to web applications, and use a simple example to demonstrate why traditional models fail to capture Ajax application behaviors.

### A. Security Threats to Web Applications

The basis of web application protection is the same-origin policy (SOP), which ensures that resources of a web application from one origin can only be accessed by JavaScript from the same origin [8]. However, if attackers manage to inject malicious JavaScript into a web application, the injected script has full access to all resources from the application's origin, causing information leakage or issuing unauthorized transactions in the victim application.




(a) A Simplified Ajax Email Application




(b) Server Side Model for the Delete Email Operation

Figure 1. A Motivating Example. *It demonstrates why traditional request-sequence-based models fail with Ajax applications.*

To inject malicious JavaScript into web applications, attackers have various attack vectors to achieve it as discussed below.

*Cross-Site Scripting (XSS) attacks [9]:* The cross-site scripting (XSS) attack is a common attack to web applications. In a typical XSS attack, the attacker exploits the sanitization vulnerabilities in a web application, and thus injects malicious JavaScript into web pages downloaded to victim users. The script injection can be implemented by crafting a URL with embedded scripts (reflected XSS), or by implanting scripts into web server's storage (persistent XSS). In either way, the injected malicious scripts will be executed in the victim web pages, and is granted with the full privileges of the victim origin. They can therefore steal and leak sensitive information, such as cookies or user inputs, or issue requests to web applications to compromise their integrity.

*Malicious web mashups:* Web mashups integrate JavaScript code from different parties into a web page, such as JavaScript libraries or advertisements. The included third-party scripts have full access to the integrator's origin. If any of these integrated third-party scripts becomes malicious in future, it can bring in malicious JavaScript into web applications.

*Network-level script injection:* A network attacker can also inject malicious scripts into non-HTTPS traffic to launch attacks in the victim user's browser. In this paper, we do not distinguish such injections from injections by XSS or web mashups, but focusing on behaviors after the script injection.

```
<script>
  function sendXHR(url, data) {
    var req = new XMLHttpRequest();
    req.open("GET", url, false);
    // more configurations on request header
    ...
    req.send(null, data);
  }

  sendXHR("http://www.email.com/listHeaders",
           "folder=inbox");
  for (var i = 0; i < 50; i++) {
    sendXHR("http://www.email.com/fetchEmail",
             "folder=inbox&index=" + i);
    sendXHR("http://www.email.com/deleteEmail",
             "folder=inbox&index=" + i);
  }
</script>
```

Figure 2. An XSS Attack Example that Attempts to Mimic a Normal Request Sequence

*Summary* Our goal is to propose a new model that captures Ajax application behaviors for attack diagnosis, regardless of the specific attack vectors used for the attacks. We show how traditional request-sequence-based models fail to capture Ajax application behaviors with the following example.

### *B. A Motivating Example*

We present a simplified Ajax email application to illustrate how a typical attack may occur in the application, and why existing request-based models fail to effectively capture the behaviors of Ajax applications.

Figure 1(a) shows the user interfaces of an Ajax-based web email application. For simplicity, suppose it has only two main tabs, *Mailbox* and *Settings*. After the user clicks on the Mailbox tab, the application will display a page for inbox, sent and trash email folders, and the user can navigate among them by clicking through the links on the left panel. Under this tab, the following requests may be sent to the server:

*http://www.email.com/sendEmail*, for sending out an email;

*http://www.email.com/deleteEmail*, for deleting an email;

*http://www.email.com/fetchEmail*, for fetching the content of an email message;

*http://www.email.com/listHeaders*, for fetching the list of email headers.

If the user clicks on the Settings tab, a page is presented to the user for preference settings, such as the email account's display name, reply-to address, email signatures, etc. The application also requests the web server for the user's existing setting preferences with *http://www.email.com/loadSettings*. When the user fills in the necessary information and clicks the *Save* button, a request, *http://www.email.com/saveSettings*, is sent to the server to update server-side account preference data.

By its design, users interact with this email system by clicking buttons or links on its web pages, which in turn sends out corresponding requests to the web server. Existing solutions model the behaviors of web applications with automata of requests [10]. For example, in order to delete an email from the user's account, the user needs to first login to the application, and by default the application opens the inbox folder and loads email headers in that folder. Then the user clicks on an email header (so the application automatically fetches its content) before clicking the delete button. The model of the above process of deleting an email is shown in Figure 1(b). If the web email application behaves normally, it is not possible to have a request of */deleteEmail* to the server without the preceding requests shown in Figure 1(b).

However, the nature of Ajax applications renders the above model ineffective in capturing the *behaviors* of Ajax applications. For example, when the web server sees a "fetch email" request followed by a "delete email" request, it is not clear what actually has happened in the client-side application. A model as in Figure 1(b) does not tell anything on whether such requests are generated from user interactions with the application or from JavaScript. In fact, malicious JavaScript in Ajax applications can easily mimic such sequences expected by the web server. Figure 2 shows such a script. Consider a compromised Ajax application *http://www.email.com* with injected malicious JavaScript. After a user logs in, the malicious JavaScript will send requests of *listHeaders*, *fetchEmail*, and *deleteEmail* in the expected sequence. This way it bypasses request-based server-side solutions, and deletes the user's first 50 emails in the Inbox folder.

From the above example, we can see that traditional server-side request-sequence-based models do not incorporate sufficient information for diagnosing attacks in Ajax applications. To compensate such insufficiency, we need to incorporate client-side state information into the model.

*Sample models:* Figure 3 presents a sneak preview of our model. As illustrated in the figure, our model includes the triggering event for the state change between the tabs of Mailbox and Settings, which represents the distinctive contextual states (represented by rectangles in the figure) where HTTP requests (represented by ovals) are generated. The model captures and abstracts the attribution between different events and actions for the execution of the email example we discuss earlier. The figure demonstrates the model captured for normal scenarios. When a mimicry attack occurs, such as the one outlined in Figure 2, the triggering events marked around the transitions between contextual states will be the script, such as "inline script, line: 32" as shown in Figure 4. Security analysts can then look up in the source code for this particular piece of script in a map to find the source code of the script. Further investigation will also reveal the origin of this script. For example, it could have

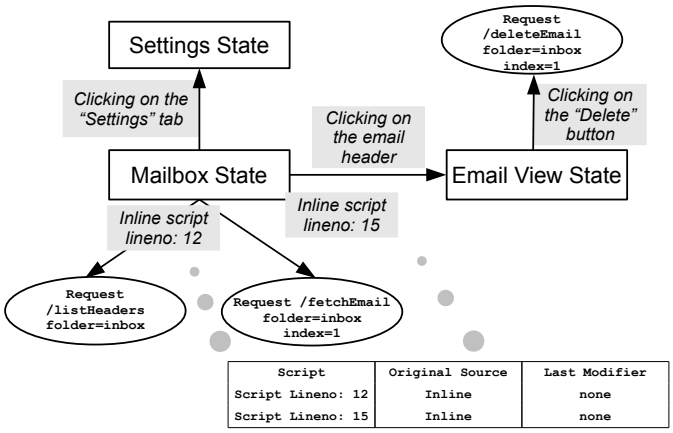



Figure 3. A Simple Sample of Our Model — Normal Scenario

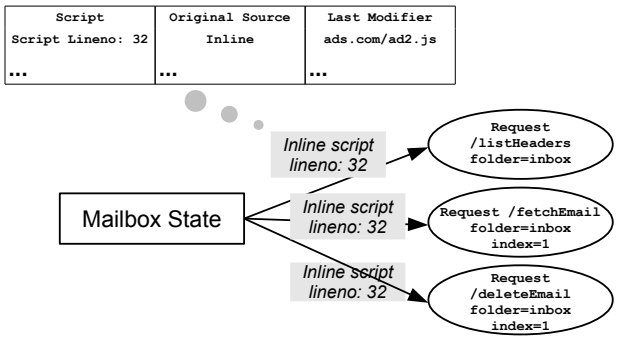



Figure 4. A Simple Sample of Our Model — Attack Diagnosis

been tampered with by an untrusted third-party JavaScript advertisement.

## III. Design

We present the definition of our model for Ajax application behaviors, and explain how the model captures the key attribution and dependency information in event-driven Ajax applications. We will show in Section V that such information is critical in diagnosing the details of attacks in Ajax applications.

### A. Modeling Ajax Web Applications

We have illustrated our model using a small example, and a formal definition of our model is presented below:

*Definition 1:* The model we propose for Ajax web applications is based on a finite state automaton (FSA) with a quintuple (S, $\Sigma$, $s_0$, $\delta$, F), where:

S is a finite, non-empty set of states. $\forall\ s = (t, sid) \in$ S, $t \in$ {APPLICATION_STATE, ACTION_STATE} and $sid$ is a string of state identifier.

$\Sigma$ is a set of input strings. $\forall\ \sigma \in \Sigma$, $\sigma$ describes an event that triggers a state transition, where $\phi \in \Sigma$ means the input string is empty.

$s_0 \in S$ is an initial state.

$\delta$ is the state transition function: $\delta$: S $\times\ \Sigma \rightarrow$ S. For $s_a, s_b \in S$ and $\sigma \in \Sigma$, $s_b$=$\delta(s_a,\sigma)$ describes the state transition from $s_a$ to $s_b$ triggered by input string $\sigma$.

- $\forall\ s_a$ = (APPLICATION_STATE, $sid_a$), $s_b$ = (ACTION_STATE, $sid_b$), if $\exists\ \delta$, s.t. $s_b = \delta(s_a, \sigma)$, then $s_a = (s_b, \phi)$.

F is the set of accepting states, and F $\subset$ S.

Definition 1 describes two types of states, APPLICATION_STATE and ACTION_STATE, for modeling web applications' client-side states and web applications' actions, respectively. Application states are abstractions of the program *contextual* states at the client side, representing the current client-side program and data states of the web application. Action states denote actions of web applications, such as invoking JavaScript functions, changing web page content, initiating particular HTTP requests, etc. In this paper, we map client-side information, such as HTTP requests and JavaScript call stack, to action states, with a focus on outgoing HTTP requests. As those actions may not necessarily change the application state, any transition from an application state to an action state is automatically paired with a reverse transition to return to the initiating application state. Each action state must be initiated from an application state as the program context for the corresponding action.

The $state$ is static information that describes the current status of the web application. The $transitions$ between the states with the specific input present the dynamic view of the web application status. Transitions among both types of states will not occur spontaneously, but must be triggered by certain events. Some transitions occur as part of browsers' standard logic, such as redirections, but sensitive operations are generally triggered by users' explicit interactions as authorizations. As a result, a substantial portion of our model is represented by the transitions between states, which precisely describe the conditions where the transitions can occur. Benefited from the full access to all client-side events in web browsers, we are able to obtain the details of all kinds of events at client-side, including user interactions on web pages, and browsers' internal events that load resources or handle asynchronous events.

Next, we elaborate more on the state and transition definitions as well as challenges we have solved in the model.

### B. Defining States in Ajax Web Application Models

In traditional desktop applications, program behaviors can be modeled using a Finite State Automata (FSA) [11], where each state is a distinct Program Counter (PC) value. However, we cannot find a straightforward way to identify similar states in Ajax applications.

One of the major differences between modeling desktop programs and modeling Ajax applications is that behaviors of desktop programs are mainly local to the system, while for

Ajax applications, both the client-side state information and its communications with the web server are crucial for behavior modeling. Client-side state information conceptually defines the context from which application behaviors occur. Each web application behavior usually occurs from a finite number of contexts. On the other hand, certain JavaScript functions and subsequent requests are the interfaces to the web application logic. Even for Ajax applications, requests are still the communication channel for client side to make any change to persistent application state stored at the server.

To comprehensively capture Ajax application behaviors, our model integrates both client-side state information and actions into states, as the two types of states.

*1) Application states: abstraction from client-side state information:* We use application states to represent the current state of the application, and the context from which actions take place. Thus, we extract various client-side state information available to web browsers to abstract the client-side information into a state identifier. Specifically, we consider the URL of the top-level document $pUrl$ and the Document Object Model structure ($DOM$) to abstract application states.

For traditional web applications, the URL of their top-level document is closely associated with a previous request that fetched the current page, so $pUrl$ would directly reflect their client-side state. However, Ajax technologies enable applications to separate the client-side content from the requests sent to the servers, and Ajax applications are not obliged to update URLs of the current document. Pages can be updated by fetching data from servers via asynchronous XMLHttpRequests and modifying the content of any element via Document Object Model interfaces. In the extreme case, however an Ajax application changes its client-side content displayed to users; its top-level URL may always remain unchanged.

Nevertheless, this caused a serious weakness of Ajax web applications, as they essentially became stateless, and users could not use their browsers' forward and backward buttons to navigate among different parts of Ajax web applications. Neither could they bookmark a particular page of Ajax applications as of traditional web sites [12].

One popular solution is to associate each state of the application with a distinct URL fragment identifier (the portion of URLs following the "#"). The client-side code of the application monitors the changes in the URL fragment identifier of the current document, and registers with browsers' history management module every URL the user visits, including the fragment identifier part. Today, this solution is widely adopted by all popular Ajax web applications as well as Ajax application frameworks such as Google Web Toolkit [13] and Yahoo! YUI Library [14]. Similarly, Google proposes using an additional exclamation mark for this purpose.

The widely used fragment identifiers in Ajax application URLs bring them back to stateful applications as traditional web applications, which also enable us to use the URL information to determine the application state.

In case URLs do not indicate application states, we need web developers to provide criteria to check part of the $DOM$ to decide the client-side application states. As such developer effort will only be needed for application states, which largely correspond to functional partitions of applications, we envision the effort to be modest.

*2) Action states:* Action states represent major important actions in web applications. The action states in our approach incorporate the destination URLs of HTTP requests, the data field if it is a POST request, and other information including the JavaScript functions on the JavaScript call stack. Our approach collects such information, and maps it to action states. For JavaScript that affects the generation of HTTP requests, our approach further tracks all influencing scripts as well, as detailed in Section III-D.

Each type of actions, as denoted by an action state, should only be triggered by particular events in particular application states. Next we explain transitions between states in our model, which carry crucial information in determining malicious behaviors.

### *C. Attributing Client-side State Transitions to Triggering Events*

The dependency between events in modern web applications provides unique strengths in diagnosing sophisticated attacks in Ajax applications. It tells whether a final request is triggered by user interactions, or from an untrusted JavaScript library. Such information helps security analysts to discern the nature of actions and trace back to the initiating party in applications with mixed contents. In our model, therefore, we mark transitions from one state to another with the triggering events at client side.

State transitions can be triggered by user interactions with web applications. For example, when a user clicks "Compose New Email" button, enters recipient, subject, and body text, and finally clicks "Send" button, a request will be sent to server containing instructions and data to send that Email. In this case, our model will have two state transitions. The first transition reflects the change of application states from the previous webpage to a page for Email composition, and is marked with the click event on "Compose New Email" button. The second transition represents an action to send a new request to the server for email sending, which is marked with the click event on "Send" button.

As we have discussed, both HTTP requests and changes to document URLs can cause state transitions, so we monitor them separately.

However, to make our model really work for flexible and powerful Ajax applications, diverse issues need to be coped with properly. In the rest of this section, we will introduce more details of our approach on how to handle different

categories of events triggering state transitions and how to extract precise attributions of asynchronous events.

### D. Diverse Categories of Triggering Events

Our approach handles different categories of events that trigger state transitions as follows:

***Resource requests*** During page loading time, the web browser parses the HTML of web pages. When it encounters `<img>` or `<script>` tags with the `src` attribute pointing to external resources, HTTP requests will be sent to fetch these kinds of resources. Similar resource requests include css background loading as well as loading favicons. However, conceptually these HTTP requests will not change the current status of Web applications, so in this case, we will create a new transition from current application state to a new action state with the category mapped by the function $f_c$, and mark this transition with the HTML tag name that triggers the HTTP request.

***URL redirection*** Web servers sometimes respond with status code 3xx [15] to redirect incoming requests to another page. This case is easy to handle in our model. We just add a new transition from current action state to a new one with the information of the new destination redirected to.

***Browsers' own functionalities*** For example, web browsers may communicate with its own or third-party websites to update its databases of malicious websites. Transitions of this category are handled similarly to resource requests, and are optimized out as attacks in out threat model would not trigger this type of requests.

***User interactions*** User interactions are the most popular types of events that trigger critical HTTP requests, so they are especially important in detecting foreign behaviors that initiate requests without users' consent. So for new HTTP requests triggered by user interactions, we mark the transition to the new action state denoting the request with the details of the triggering user interactions, for example, a click on a button named "settings".

***JavaScript*** JavaScript in web applications can directly issue HTTP requests, or generate simulated user interactions that in turn trigger HTTP requests. We record all such behaviors in our model. As illustrated earlier in Figure 4, for each JavaScript (or timer as we explain below), we track the source origin of the script and the origins of all other scripts that modify it. For brevity, we record only the last modifying script in this paper. The details of creator and modifier information for scripts accompany the model constructed.

***Timers*** Another category of client-side events triggering transitions of states are timers. Web browsers have internal timers as part of their own implementation, and web applications can also use JavaScript to create timers to execute certain script code after a period of time, or at every certain interval. As timers themselves are not really the source of the script code they trigger, in our model we trace back to the creators of timers. For timer-triggered transitions to new action states, we attribute them to the initial creators of timers that finally trigger state transitions, for example, a particular `script` tag.

### E. Identifying Dependency between Asynchronous Events

To extract the attributions of HTTP requests, we need to maintain the precise dependency between asynchronous events, such as user interactions, timers, XMLHttpRequest callbacks, etc. Such dependency cannot be based on the chronological relations, which may vary from run to run.

Instead, we build a structure called virtual stack that simulates the behavior of call stacks generated by debuggers. In web browsers, events are handled by different functions or by the same function with different arguments. We push onto virtual stack the event information whenever the corresponding function with relevant arguments starts to execute, and pop it out just before the function returns. This structure is by nature accurate in deciding the attribution of events as long as the corresponding functions are included in the virtual stack mechanism. To enable additional attribution among events across different call stack instances, we also use the objects shared by different functions or events as targets to associate relevant client-side events.

For dynamically generated JavaScript, we track the original script that generates it and the last script that modifies it; for timers set up by web application script, we similarly uses the original script that sets up the timer and the last script that modifies it as the attributing objects. For these cases, if an HTTP request is generated as a result, we use the event(s) that triggers the execution of the original script as the triggering event for the request. For example, if the original script is a click event listener function, the triggering event will be the click event and the JavaScript event listener function.

### F. Model Construction & Attack Diagnosis

The proposed model of Ajax applications is constructed automatically by an in-browser monitoring system. During the web sessions when users or security analysts interact with the web applications, our system records the internal events occurring in the browser, and establishes the attribution and dependency between them. Along with the execution of Ajax applications, our system builds up the behavior models with the event dependency information as we explain earlier.

To diagnose attacks with such models, security analysts can optionally run our system in a trusted environment to obtain models for normal application behaviors. When a security incident occurs and reported, security analysts can retrieve the behavior models recorded for the corresponding exploited session or date. Although in general it is hard to search for attack points in the models, in practice, security analysts can either compare them with normal behavior

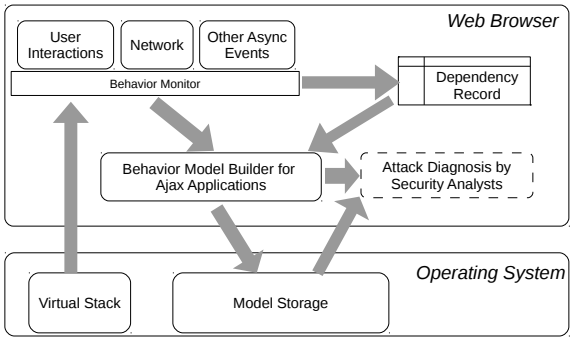



Figure 5. Architecture Overview

models or use reported attack information, such as the URLs and parameters of malicious HTTP requests, to search in the models. This way, our models help them understand the causes and vectors of the attacks, which may be helpful in proposing a quick fix to the vulnerability.

## IV. Implementation

We implemented a proof-of-concept prototype of our solution in Firefox 3.5 and 8.0. Our implementation is an extension to a security framework for web browsers [16]. Figure 5 illustrates the architecture overview of our solution. The Behavior Monitor is a run-time module in the web browser, which monitors user interactions, network requests, and asynchronous events generated by web applications or the browser itself. It then extracts necessary information of these events or actions, obtains information from the Virtual Stack and build the dependency between them. Such dependency is recorded into Dependency Record. During the execution of Ajax applications, the Behavior Model Builder constructs the models according to the information monitored by the Behavior Monitor and stores them in the file system.

In our Firefox-based prototype, the interception of asynchronous events and actions are achieved by hooking relevant functions in classes such as `nsEventListenerManager`, `nsWindow`, `nsWebShell`, `nsDocShell`, `nsHttpChannel`, `nsHTMLInputElement`, `nsXMLHttpRequest`, `nsImageLoadingContent`, `nsFrameLoader`, `nsScriptLoader`, `nsFormSubmission`, `nsCSSValue`, `nsGlobalWindow`, etc. Our approach is implemented in C/C++, using libraries such as C++ STL to manipulate strings and hash maps, and Boost libraries to perform serialization and deserialization of our models constructed between the memory and the hard disk.

Our core model, including the model itself, its basic operations, the Virtual Stack and the Dependency Record, is implemented in 1K SLOC C++ code. The functions handling browser events, actions as well as invoking operations on the model, counts around 4.4K SLOC C code.

## V. Evaluation

In order to evaluate the effectiveness of our solution in modeling Ajax applications, we deploy our solution to model the behaviors of open-source and popular Ajax web applications, and demonstrate its capability in diagnosing malicious behaviors with simulated code injection attacks on those Ajax applications. In general, attackers can leverage various vectors to introduce malicious scripts into web applications, including exploiting their sanitization vulnerabilities, compromising mashup content included by them, or by setting up a malicious proxy. As our solution is independent of the actual attack mechanisms, in our case studies, we use browser extensions to inject scripts into web applications.

### A. Case Studies: Diagnosing Injection Attacks

We deployed our solution to evaluate the effectiveness of malicious behavior diagnosis with an open-source Ajax webmail application, Claros inTouch [17] and Twitter [18].

We simulated the mapping functions from URLs to application and action states to facilitate our testing. For Claros inTouch, we simulated the developer effort in mapping 7 URLs to 7 application states. This effort is trivial, and only requires basic understanding of the application functionality, such as *inbox*, *compose*, *settings*, etc. After manual browsing with most of its functionalities, our system constructed a model of 7 application states and 54 action states, and 203 different transitions between states. As for Twitter, our system leveraged the fragment identifiers in its URLs to obtain a behavior model of 30 application states, 185 action states, and 644 transitions between states.

To verify the effectiveness of our solution, we tested with four injection attacks.

*Attacks on inTouch:* The first attack attempts to delete an email without the user's consent, similar to the one in Figure 2. The injected scripts automatically send an Ajax request to delete a user email.

Figure 6 shows a fragment of an example model constructed for the inTouch webmail application. In the figure, rectangle states are *application states*, transitions from whom require specific user actions as marked along the arrowed lines. Oval states denote *action states* for requests that automatically transit back to application states after the requests are sent. Dashed lines indicate state transitions triggered by user actions, while dotted lines denote automatic state transitions from action states to stable states.

This example model fragment leverages users' interactions with the web application, and specifically includes information on how state transitions can happen, and requests sent to the server at each state, transition from "State 2 Settings" to "Request */deleteEmail*", marked as "inline script lineno: 78", which was last modified by a browser extension. The model shown in Figure 6 clearly illustrates the malicious

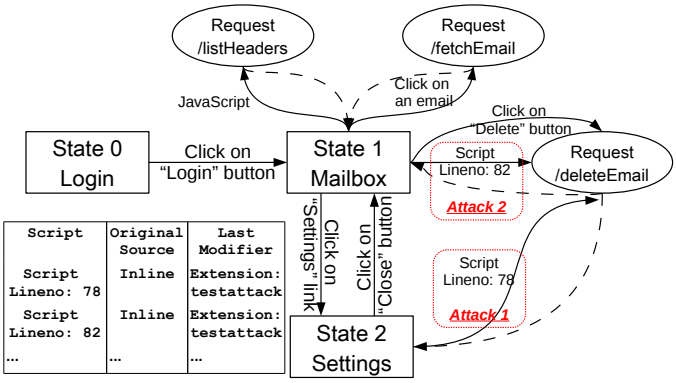



Figure 6. Example Model Fragment for inTouch *Attacks marked in dashed rectangles.*

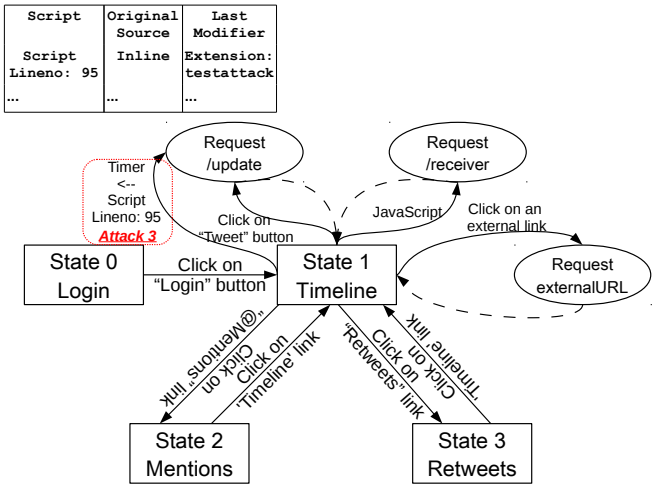



Figure 7. Example Model Fragment for Twitter *Attack marked in dashed rectangles.*

request sent from an unexpected application state, triggered by a script modified by a browser extension.

In the second case, the malicious script was injected into the "Mailbox" page to send the email deletion request (*Attack 2*). Thus, the model constructed for this scenario contains a transition from "State 2 Mailbox" to "Request */deleteEmail*". Although such a state transition also exists with normal behaviors, they differ in the triggering events. In normal cases, such an email deletion request is triggered by a user click on the "Delete" button, while in our experimental scenario, it was triggered by an inject script. Such information can help security analysts to discern even carefully crafted mimicry attacks.

*Attack on Twitter:* Similarly, Figure 7 illustrates a fragment of a sample model for the behaviors of Twitter. We simulated a malicious script that posted a tweet secretly without the user's awareness (*Attack 3*). Twitter has a more sophisticated authentication mechanism that requires the requests posting tweets to contain certain HTTP headers as well as a per-session secret token value as HTTP POST data. Thus, we designed a more sophisticated attack script accordingly. The injected script used prototype poisoning to intercept calls to XMLHttpRequest. It also searched the page content for the secret token value in a hidden HTML Input element. Then it set up a timer with `setInterval` to wait for a few seconds, so that Twitter called an `XMLHttpRequest` to send normal requests to keep the session live. Such calls were intercepted by the malicious script, which learned all HTTP header values. Then the malicious script composed a malicious request with the correct HTTP values as well as the secret token in the POST data, to post a new tweet automatically.

This attack example is rather stealthy and sophisticated. However, since the malicious request was generated by a JavaScript timer, our model still captured this malicious behavior that is not seen in normal tweeting scenarios. It differed from normal tweet-posting requests that are triggered only when a user clicks on the "Tweet" button.

*Diagnosing simulated clicks:* JavaScript code can simulate user clicks by invoking `document.CreateEvent("MouseEvent")` and dispatch it to any element in the web page. We compose one more attack script that automatically fills in the "Compose new Tweet" input box on Twitter page, and simulates a click on the "Tweet" button (*Attack 4*). Our system captures a fragment of behavior model shown in Figure 8. The model constructed by our solution marks the triggering event for the tweet-posting request as a combination of the user click and the script that has actually simulated the click. This tells the exact details of the steps of the attack, without confusing security analysts between simulated and real user clicks.

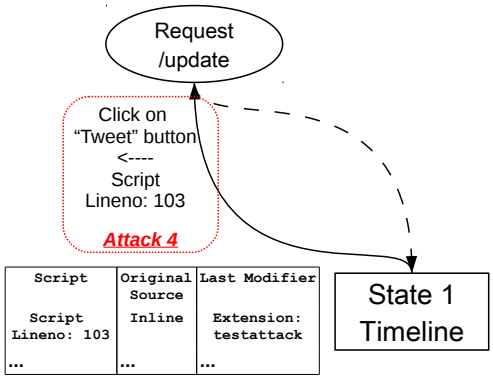



Figure 8. Model for Simulated Click on the "Tweet" Button *Attack marked in dashed rectangles.*

*Summary* With these attack examples, we demonstrate the importance of client-side event dependency in attack diag-

| Web Sites | Original Firefox | Our Prototype (Overhead) |
|---|---|---|
| Google | 0.3835 | 0.4601 (19.97%) |
| Facebook | 3.535 | 4.169 (17.93%) |
| YouTube | 2.994 | 3.975 (32.77%) |
| Yahoo | 3.234 | 3.592 (11.07%) |
| Baidu | 3.667 | 4.167 (13.64%) |
| Wikipedia | 6.739 | 6.850 (1.65%) |
| Blogger | 2.079 | 2.113 (1.64%) |
| Windows Live | 2.735 | 3.029 (10.75%) |
| Twitter | 6.258 | 7.383 (17.98%) |
| Amazon | 4.412 | 4.854 (10.02%) |

Table I
AVERAGE PAGE LOAD TIME ON 10 RUNS, IN SECONDS

nosis for Ajax applications. It confirms that user interaction details are crucial in investigating malicious behaviors in web applications.

### B. Performance Overhead

To evaluate the performance overhead of our prototype system, we measured the average page load of the Alexa Top 10 web sites, which is a macrobenchmark on the processing slowdown of the browser with our solution. Our experiments were performed on an Ubuntu 11.10 32-bit desktop machine with Intel®Core™2 Duo CPU E6550 @ 2.33GHz and 4GB memory. Our results demonstrated around 10-30% of performance overhead, which is reasonable for an analysis tool.

Our evaluation compared the page load time with the original Firefox 8.0 browser and with our instrumented browser, and the result is shown in Table I. Instead of the simple first pages, we logged into all web applications that require authentication, such as Facebook, Blogger, Twitter, etc., while the exception was for Windows Live, as its pages after login are incompatible with our browser testbed.

### C. Discussion on Mappings from URLs to Action States

We assume web developers will provide mapping functions from URLs to action states, which means such mapping tells our model of the category and purpose of each HTTP requests. During our evaluation, we manually created less than 20 simple regular expressions according to our experience and observation. A better way to handle this issue is to develop or derive certain sets of heuristics for each web application in a full- or semi-automatic way. This is possible as many web applications share similar patterns and schemes in forming their URLs.

## VI. RELATED WORK

### A. Applications of Program Behavior Models

Sekar et al. [11] proposed a finite state automaton (FSA) based intrusion detection system (IDS). The model proposed in their work defines state in FSA as distinct program counter (PC) values at the point of system call invocations, while transitions as corresponding system calls. It is deterministic during detection phase, thus affordable in running time. Feng et al.'s work [19] supplemented [11] by incorporating return address information in call stacks, and generating abstract execution paths between two execution points. The additional call stack information considered enables this method to detect some attacks missed by previous ones, like impossible path exploits.

IDS can also be deployed on network [20]–[22] to effectively monitor network traffic. Similarly, with the popularity of web applications and web-based attacks, network IDS solutions are also adapted to the new web environment. Pioneering work in this area [1]–[3] started to adapt network IDS to monitor the behavioral state of web clients. These research works share the same basic idea with us: leveraging the power of anomaly IDS to detect web-based attacks. However, they are not oriented to Ajax applications, and do not observe Ajax application characteristics such as asynchronous HTTP requests and user actions.

Guha et al. [10] proposed an approach that is based on static analysis of client-side web application source code, including HTML and JavaScript, which generates a non-deterministic request graph. This graph will be used as normal behavior constraints to detect runtime anomalies. One major insufficiency of this solution is static analysis cannot cover all dynamics of JavaScript language, so the protection is not complete. Another issue is that although this paper highlights the new challenges Ajax applications, i.e., asynchronous events and HTTP request, and dynamically generated JavaScript code, it does not provide a complete solution to these issues.

The dynamic nature of Ajax web applications has also attracted researchers to tackle their challenges to other areas. For example, Crawljax [4] proposes new frameworks to crawl the pages of Ajax web applications, which enables automatic software testing for Ajax applications [5]–[7]. Similarly, AjaxTracker [23] develops an automatic tool to mimic human interactions to measure Ajax applications to better understand their characteristics. Other works attempt to identify program bugs by comparing the executions of web applications in different browsers [24], [25]. The idea of testing web applications by fuzzing user inputs has also been examined by McAllister et al. [26]. Our solution differs with them in focusing on handling Ajax application challenges to diagnose the behaviors of attacks in these applications.

On the other hand, there are also several formal frameworks [27]–[29] for modeling asynchronous complex systems. Our goal in this work differs with them. Instead of capturing system behaviors and semantics in a holistic manner, we are proposing a simple yet practical run-time model that captures particular aspects that are critical in security analysis: event contexts and dependency.

### B. Detection and Prevention of Web-based Attacks

There has been extensive work on addressing security threats in web applications. For example, Yue et al. [30] instrument Web browsers to obtain JavaScript execution traces and performed offline analysis on these traces. Similarly, Oystein et al. [31] propose a solution to audit JavaScript code executions and analyze the traces using misuse IDS, based on known signatures. Noxes [32] acts as a web proxy to filter network connections of web applications with policies generated automatically or specified manually. Other works like [33]–[35] rewrite JavaScript code to make it more secure or self-protecting.

On the other hand, server side solutions tend to automatically find vulnerabilities in web applications by code analysis [36] and/or runtime checking [37]–[39]. The work of [37] combines static and dynamic techniques. By monitoring information flow, a lattice-based static analysis is used to identify sections of code with potential vulnerabilities. Then these sections of code are instrumented with runtime security guards.

Some other research collaborates the server and client sides of web applications to detect and prevent web-based attacks, such as BLUEPRINT [40] and Document Structure Integrity (DSI) [41].

Our work in this paper focuses on analyzing the behaviors of attacks on Ajax applications. It can be used in conjunction with existing defenses solutions on web-based attacks, and potentially improve the accuracy in attack detection.

## VII. Conclusion

The event-driven, asynchronous Ajax applications pose new challenges for modeling application behaviors, as required by diagnosis of security incidents in Ajax applications. In this paper, we propose a new behavior model for Ajax applications. The model captures the *contexts* where actions take place as well as attribution and dependency between events. The proposed model provides the insightful details on the root cause and developments of attacks in Ajax applications. Our in-browser behavior monitoring system automatically constructs the behaviors models for security analysts to examine. We implement a prototype of our solution in Firefox, and demonstrate its effectiveness of attack diagnosis with case studies on real-world web applications.

## Acknowledgements

We thank the anonymous reviewers for their insightful comments for improving this paper. We also thank Xuhui Liu and Utsav Saraf for their feedback on this work. We had interesting discussions with Zhenchang Xing and Manchun Zheng on our ideas, which also referred us to a few related works. This work is supported in part by Singapore Ministry of Education's grant R-252-000-460-112.

## References

[1] C. Kruegel and G. Vigna, "Anomaly detection of web-based attacks," in *Proceedings of the 10th ACM Conference on Computer and Communications Security (CCS)*, 2003.

[2] K. Jayaraman, G. Lewandowski, P. G. Talaga, and S. J. Chapin, "Enforcing request integrity in web applications," in *Proceedings of the 24th Annual IFIP WG 11.3 Working Conference on Data and Applications Security and Privacy (DBSec)*, 2010.

[3] C. Chriscione and S. Zanero, "Masibty: an anomaly based intrusion prevention system for web applications," http://www.blackhat.com/presentations/bh-europe-09/Zanero_Criscione/BlackHat-Europe-2009-Zanero-Criscione-Masibty-Web-App-Firewall-wp.pdf, 2009.

[4] A. Mesbah, E. Bozdag, and A. v. Deursen, "Crawling Ajax by inferring user interface state changes," in *Proceedings of the 8th International Conference on Web Engineering (ICWE)*, 2008.

[5] A. Mesbah and A. van Deursen, "Invariant-based automatic testing of Ajax user interfaces," in *Proceedings of the 31st International Conference on Software Engineering (ICSE)*, 2009.

[6] C. Bezemer, A. Mesbah, and A. van Deursen, "Automated security testing of web widget interactions," in *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2009.

[7] D. Roest, A. Mesbah, and A. v. Deursen, "Regression testing Ajax applications: Coping with dynamism," in *Proceedings of the Third International Conference on Software Testing, Verification and Validation (ICST)*, 2010.

[8] Wikipedia, "Same origin policy," http://en.wikipedia.org/wiki/Same_origin_policy.

[9] ——, "Cross-site scripting," https://en.wikipedia.org/wiki/Cross-site_scripting.

[10] A. Guha, S. Krishnamurthi, and T. Jim, "Using static analysis for Ajax intrusion detection," in *Proceedings of the 18th International Conference on World Wide Web (WWW)*, 2009.

[11] R. Sekar, M. Bendre, D. Dhurjati, and P. Bollineni, "A fast automaton-based method for detecting anomalous program behaviors," in *Proceedings of the 2001 IEEE Symposium on Security and Privacy*, 2001.

[12] Wikipedia, "Ajax (programming)," http://en.wikipedia.org/wiki/Ajax_(programming), 2010.

[13] G. W. Toolkit, "Coding basics - history," http://code.google.com/webtoolkit/doc/latest/DevGuideCodingBasicsHistory.html.

[14] Yahoo, "YUI 2: Browser history manager," http://developer.yahoo.com/yui/history/.

[15] W3C, "Redirection 3xx, status code definitions," http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html#sec10.3.

[16] X. Dong, K. Patil, X. Liu, J. Mao, and Z. Liang, "An entensible security framework in web browsers," Systems Security Group, School of Computing, National University of Singapore, Tech. Rep. TR-2012-001, Feburary 2012.

[17] Claros, "inTouch," http://www.claros.org/web/showProduct.do?id=1.

[18] "Twitter," http://twitter.com.

[19] H. H. Feng, O. M. Kolesnikov, P. Fogla, W. Lee, and W. Gong, "Anomaly detection using call stack information," in *Proceedings of the 2003 IEEE Symposium on Security and Privacy*, 2003.

[20] R. Sommer and V. Paxson, "Enhancing byte-level network intrusion detection signatures with context," in *Proceedings of the 10th ACM Conference on Computer and Communications Security (CCS)*, 2003.

[21] C. Gates and C. Taylor, "Challenging the anomaly detection paradigm: a provocative discussion," in *Proceedings of the 2006 Workshop on New Security Paradigms (NSPW)*, 2007.

[22] M. A. Aydın, A. H. Zaim, and K. G. Ceylan, "A hybrid intrusion detection system design for computer network security," *Computers and Electrical Engineering*, vol. 35, no. 3, 2009.

[23] M. Lee, R. R. Kompella, and S. Singh, "Ajaxtracker: active measurement system for high-fidelity characterization of Ajax applications," in *Proceedings of the 2010 USENIX Conference on Web Application Development (WebApps)*, 2010.

[24] S. Roy Choudhary, H. Versee, and A. Orso, "Webdiff: Automated identification of cross-browser issues in web applications," in *Proceedings of the 2010 IEEE International Conference on Software Maintenance (ICSM)*, 2010.

[25] A. Mesbah and M. R. Prasad, "Automated cross-browser compatibility testing," in *Proceedings of the 33rd International Conference on Software Engineering (ICSE)*, 2011.

[26] S. Mcallister, E. Kirda, and C. Kruegel, "Leveraging user interactions for in-depth testing of web applications," in *Proceedings of the 11th International Symposium on Recent Advances in Intrusion Detection (RAID)*, 2008.

[27] R. Milner, *Communicating and mobile systems: the $\pi$-calculus*. Cambridge University Press, 1999.

[28] C. A. R. Hoare, *Communicating sequential processes*. Prentice-Hall, Inc., 1985.

[29] OASIS, "Oasis web services business process execution language (wsbpel) tc," https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsbpel.

[30] C. Yue and H. Wang, "Characterizing insecure javascript practices on the web," in *Proceedings of the 18th International Conference on World Wide Web (WWW)*, 2009.

[31] O. Hallaraker and G. Vigna, "Detecting malicious javascript code in mozilla," in *Proceedings of the 10th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS)*, 2005.

[32] E. Kirda, C. Kruegel, G. Vigna, and N. Jovanovic, "Noxes: a client-side solution for mitigating cross-site scripting attacks," in *Proceedings of the 2006 ACM Symposium on Applied Computing (SAC)*, 2006.

[33] D. Yu, A. Chander, N. Islam, and I. Serikov, "Javascript instrumentation for browser security," in *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2007.

[34] Google, "Caja," http://code.google.com/p/google-caja/.

[35] S. Isaacs and D. Manolescu, "WebSandbox - Microsoft Live Labs," http://websandbox.livelabs.com/, 2009.

[36] G. Wassermann and Z. Su, "Static detection of cross-site scripting vulnerabilities," in *Proceedings of the 30th International Conference on Software Engineering (ICSE)*, 2008.

[37] Y. Huang, F. Yu, C. Hang, C. Tsai, D. Lee, and S. Kuo, "Securing web application code by static analysis and runtime protection," in *Proceedings of the 13th International Conference on World Wide Web (WWW)*, 2004.

[38] S. Nanda, L. Lam, and T. Chiueh, "Dynamic multi-process information flow tracking for web application security," in *Proceedings of the 2007 ACM/IFIP/USENIX International Conference on Middleware Companion (MC)*, 2007.

[39] G. A. D. Lucca, A. R. Fasolino, M. Mastoianni, and P. Tramontana, "Identifying cross site scripting vulnerabilities in web applications," in *Proceedings of the Web Site Evolution, Sixth IEEE International Workshop (WSE)*, 2004.

[40] M. Ter Louw and V. N. Venkatakrishnan, "Blueprint: Robust prevention of cross-site scripting attacks for existing browsers," in *Proceedings of the 2009 IEEE Symposium on Security and Privacy*, 2009.

[41] Y. Nadji, P. Saxena, and D. Song, "Document structure integrity: A robust basis for cross-site scripting defense," in *Proceedings of the 16th Annual Network and Distributed System Security Symposium (NDSS)*, 2009.