Solving Some Modeling Challenges when Testing Rich Internet Applications for Security

Suryakant Choudhary[1], Mustafa Emre Dincturk[1], Gregor v. Bochmann[1,3], Guy-Vincent Jourdan[1,3]
[1]EECS, University of Ottawa
[3]IBM Canada CAS Research
Ottawa, Canada
{schou062, mdinc075}@uottawa.ca
{bochmann, gvj}@eecs.uottawa.ca

Iosif Viorel Onut, Paul Ionescu
Research and Development, IBM
IBM® Security AppScan® Enterprise
Ottawa, Canada
{vioonut, pionescu}@ca.ibm.com

Abstract—**Crawling is a necessary step for testing web applications for security. An important concept that impacts the efficiency of crawling is state equivalence. This paper proposes two techniques to improve any state equivalence mechanism. The first technique detects parts of the pages that are unimportant for crawling. The second technique helps identifying session parameters. We also present a summary of our research on crawling techniques for the new generation of web applications, so-called Rich Internet Applications (RIAs). RIAs present new security and crawling challenges that cannot be addressed by traditional techniques. Solving these issues is a must if we want to continue benefitting from automated tools for testing web applications.**

Keywords: Security Testing, Automated Crawling, Rich Internet Applications, State Equivalence

I. Introduction

The concerns on the security of the web applications have grown along with their popularity. One of the response to these concern about security issues was the development of automated tools for testing web applications for security.

There are various commercial and open-source black-box web application security scanners available (see [1] for a recent survey). A black-box web application security scanner is a tool that aims at finding security vulnerabilities in web applications without accessing the source-code. That is, a black-box scanner has only access to the client-side just like a regular user of the application. When a black-box security scanner is given the URL pointing to the initial page of the application (together with other minimal information that may be required, such as username and password, if the application requires login), it simply tries to discover every page (client-state) of the application that is reachable from the initial page. As the new pages are discovered, the tool scans each one for possible security vulnerabilities by applying test cases and reports any detected vulnerability to the user. These tools can easily apply a large number of security tests automatically at each discovered page which would otherwise require a long time if done manually.

It is clear that effectiveness of a security scanner depends not only on the quality and coverage of the test cases but also on how efficient it is at discovering the pages (client-states) of the application. This activity of automatic exploration of the web application is called crawling. The result of crawling is called a "model" of the application. This model simply contains the discovered client-states and ways to move from one state to the other within the application. Only after obtaining a model of the application can a security scanner know which states exists and how to reach them in order to apply the security tests. Thus, crawling is a necessary step for security scanning as well as it is necessary for content indexing and testing for any other purpose such as accessibility.

The new generation of web applications, sometimes called Rich Internet Applications (RIAs) poses a challenge for the security scanners. This is because, the crawling techniques used for traditional web applications are not sufficient for RIAs. Without an appropriate crawling ability a security scanner cannot be used on RIAs.

RIAs are much more responsive and user-friendly than their traditional counterparts. This is thanks to technologies such as AJAX (Asynchronous JavaScript and XML) [2] which combine client-side scripting with asynchronous communication. That is, client-side scripts (JavaScript) allow computations to be carried out at the client-side. It is also possible to register JavaScript code as event handlers on HTML elements so that when a user interacts with the element the corresponding event (click, mouse over etc.) fires and the registered code is run. When JavaScript code runs, it has the capability of accessing and modifying the current page by changing the DOM (Document Object Model) [3] which represents the client-state of the application. In addition, these scripts can communicate asynchronously with the server so that new content from the server can be retrieved and used to modify the current state into a new one. In a sense, RIAs have brought the feeling of the desktop applications to the web while still preserving the convenience of being on the web. As a result, there are RIA versions of even the most classical desktop applications (word processors, photo editors, media players etc.)

Traditional crawling techniques are based on the fact that in a traditional application each page is identified by a URL. So, to crawl a traditional application it is enough to collect and visit the URLs on the each page to discover the states of the application. But in RIAs, client-side scripts are able to change the state on the client-side, which means that the URL does not necessarily change when the state changes (many RIAs have a single URL which points to the initial state). That means that in order to crawl RIAs, a crawler needs to exercise event-based exploration as well as the traditional URL-based exploration.

Primarily motivated by the aim of making security scanners usable on RIAs, our research group has been working in collaboration with IBM to design efficient RIA crawling techniques. We have implemented some of the ideas of this research in a prototype of IBM® Security AppScan® Enterprise [4], a security scanner for web applications.

Previously we have presented in detail the requirements and anticipated problems for a solution to the RIA crawling problem as well as the shortcomings of the current attempts to solve this problem [5]. Basically we require a RIA crawler to 1) produce a complete model in a deterministic way (it should capture all the states and be able to produce the same model when run multiple times on an unchanged website); 2) be efficient (discover as many states as possible in a given amount of time); 3) use an appropriate state equivalence relation (i.e. a mechanism to identify the states that should be considered as the same) depending on the purpose of the crawl and the application.

We have also stated that satisfying these requirements may be possible only when some limiting, but common assumptions, are made about the application: 1) the application should not have server-side states. That is, the global state of the application is only determined by the client-state. In that case we can expect that repeating an action on the same client state at different times will always result in the same state. 2) On a page where the user can enter a free text input in a form, using a finite set of representative input values is enough for crawling purposes. This last assumption helps us not to miss any state due to not entering a specific user-input value, since in practice the result of an action might be different based-on the user-input values.

Some of the important anticipated problems were designing efficient strategies, choosing appropriate user-input values, deciding what to do if the application has some server-side states, how to choose an appropriate state equivalence relation, how to deal with the state explosion problem and how to further enhance the model for security.

As a step forward in addressing some of these issues, in this paper we focus on two important concepts that are important for efficient RIA crawling: state equivalence and crawling strategy. State equivalence is the mechanism used by the crawler to decide whether or not two states should be regarded as the same. This is important since there are often situations where, although two pages reached by the crawler are not exactly the same, they are equivalent for the purpose of crawling. The simple example is an application that places an advertisement in each page. For this application, visiting the same page at different times will most likely result in pages that have different advertisements while everything else remains the same. Although these two pages are not exactly the same, a good state equivalence relation should detect and ignore these "unimportant" parts. Failure to do so will likely result in infinite crawling runs or at least unnecessarily large state spaces.

A crawling strategy is an algorithm which decides what actions to take next. For example, the crawling strategy decides which URL to follow next if it is doing traditional URL-based exploration and in the case of event-based exploration (in RIAs), the crawling strategy decides from which state which event should be executed next.

We have been working on the techniques to improve state equivalence relations in addition to designing crawling strategies for event-based crawling of RIAs. In this paper, we present two techniques that help to improve the existing state equivalence relations. Also we present a summary of our research on crawling strategies.

This paper is organized as follows. In Section 2, we present the techniques for improving any state equivalence relation. Section 3 contains a summary of our recent research on crawling strategies for RIAs. Section 4 presents the related work and Section 5 concludes with a summary of contributions.

II. State Equivalence

During crawling, the crawler navigates through the states of the application by following the found links or executing an event on the current state. When the crawler reaches a state, it must know if it had already been to that state before. Otherwise, the crawler would regard each state as a new one and it could never finish crawling and build a meaningful model, it would just keep exploring the same states over and over again.

The mechanism that is used by the crawler to decide if a state is equivalent to an already discovered one is called the state equivalence relation. With a state equivalence relation the crawler will recognize an already visited state and hence it will not explore already explored events unnecessarily. Moreover, by identifying the current state it will know where it currently is in the partial model constructed up to that point. Thus, the crawling strategy will be able to decide on its next moves based on that model (i.e. crawling strategy will know whether there is an already executed sequence of events that will lead from the current state to some other state).

It seems very difficult to come up with a single solution for equivalence. The simplest equivalence relation is the equality. In this case, two states are considered equivalent only if they are identical. But for most applications, this definition is too strict and would lead to very large or infinite models. Clearly, deciding whether a given application state is "similar" to another state very much depends on the application as well as the purpose of the crawl. As an example, if the purpose of crawling is security scanning then the text content of the pages would not be very important, hence could be ignored. However, from the content indexing and usability point of view, the text content should not be ignored when deciding the equivalence of states. Being able to adapt the equivalence relation to the application and the purpose of the crawl is thus very valuable. For these reasons, we believe that state equivalence should be considered independent of the crawling strategy.

The choice of an appropriate equivalence relation should be considered very carefully. If an equivalence evaluation method is too stringent (like equality), then it may result in too many states being produced, essentially resulting in state explosion, long runs and in some cases infinite runs. On the contrary, if the equivalence relation is too lax, we may end up with client states that are merged together while, in reality, they are different, leading to an incomplete, simplified model.

Although choosing a state equivalence relation requires many considerations, its correctness should not be put into negotiation. That is, a state equivalence relation should be an equivalence relation in mathematical sense. Moreover, it seems reasonable to insist that equivalent states have the same set of events, since otherwise two equivalent states would have different ways to leave them.

In the following, we present two novel ideas to help improve the efficiency of state equivalence of web application being crawled in an automated and efficient manner.

A. Load, Reload: Discovering Unnecessary Dynamic Content of Web Page.

As mentioned above, Web pages often contain bits of content that change very often but are not important in terms of making two states non-equivalent. When determining whether or not two states are equivalent, there is a desire to be able to ignore these constantly changing but irrelevant portions of the page. This is important since failing to identify data that should be ignored could cause an equivalence function to evaluate to false when it otherwise would not.

Thus one of the important challenges when defining state equivalence functions is to exclude from the content considered in the equivalence function the portion of the page/DOM that may introduce false positives. The most common current solution to the problem is to manually configure the crawler on a case by case basis, to make it ignore certain types of objects that are known to change over time, such as session ids and cookies. This is highly inefficient, and is also inaccurate, since most of the time this list is incomplete. Another solution is to use regular expressions to identify in the DOM the portions of the content that can be ignored. The main problem with the latter solution is the difficulty of creating the regular expressions and the fact that they are different for different sites. Automating the detection of irrelevant page sections is desired since those differences are also page-specific and as a consequence, the irrelevant parts vary from page to page even within the same website.

We have developed a technique for automatically inferring the portions of the page that should be ignored. The technique requires loading a given web page twice. The DOM of the page at each load can then be compared to see the differences which indicate data that can be ignored. For example, a web page X is loaded at time t_1 and then again at time t_2. The DOM of X at t_1 is then compared to the DOM of X at t_2 to produce Delta(X), in the form of a list of differences between the DOMs. When using an equivalence function to compare this state with another, the data in this list can be excluded. Therefore, two states can be considered equivalent if they are equivalent after the irrelevant data is excluded from both.

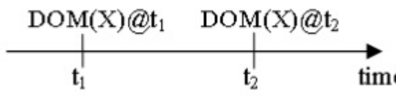
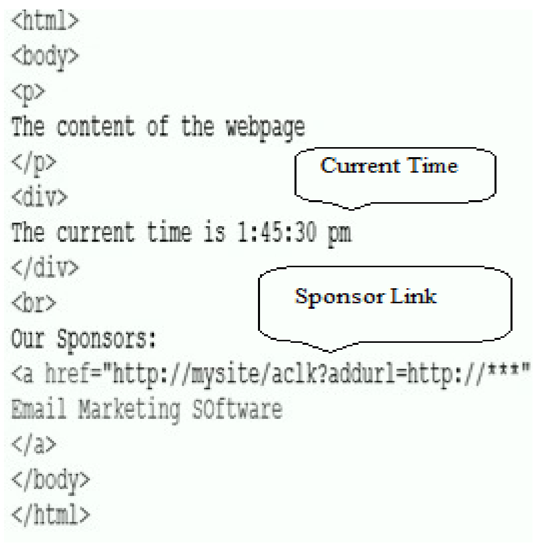


Figure 1. DOM values for a web page X at two different time intervals

Figure 1 depicts a timeline with t_1 and t_2 being two distinct points in time. Assume that the crawler reaches and loads a web page X at time t_1 producing DOM(X) @t_1. The same web page X loaded at time t_2 will produce DOM(X) @t_2. Let Delta(X) represent the differences between DOM(X) @t_1 and DOM(X) @t_2. Delta(X) is the information that must be excluded by the DOM equivalence function for web page X. This delta can be computed as a string difference between the two DOMs, or can be pictured as a collection of XPath values. Each of the XPath values will point to an element/location of the DOM that can be ignored. Furthermore, if an attribute value is different between two DOMs, the XPath will point not only to the node, but also to that attribute within the node that is not consistent in time.

The crawler will then record Delta(X) as being the irrelevant information to be excluded for any future DOM comparisons for the web page X.



```
<html>
<body>
<p>
The content of the webpage
</p>
<div>
The current time is 1:45:30 pm
</div>
<br>
Our Sponsors:
<a href="http://mysite/aclk?addurl=http://***"
Email Marketing SOftware
</a>
</body>
</html>
```

Figure 2. Example of a page with irrelevant data which changes over time

Let us analyze a simple example of a page X that, after rendering, contains the HTML document shown in Figure 2. As we can see, the webpage is displaying the current time, and let us assume it displays one random sponsor at a time. Let DOM(X) @t_1 be the document in Figure 2. Thus, it is feasible to assume that a different request to the same server for page X will return a different timestamp and a different sponsor.

Furthermore, let us assume that the second time when page X is visited, the content will change as follows:

Current time will be “1:45:31 pm” and the sponsor will be “http://mysite/aclk?sa=l&ai=Ba4&adurl=http://www.mysite”

We can now compute the Delta(X) as being the list of differences as follows:

Delta(X) = {html\body\div\, html\body\a\@href}

There are many ways to exclude Delta(X) from the DOM. For instance, each XPath that exists in the Delta(X) can be simply deleted from the DOM. Alternatively, every XPath from Delta(X) can be replaced with any of the t_1 or t_2 values. The idea behind is to have these two values equal after the current step. This will make the DOM comparison algorithm to see all the XPath in the Delta(X) as having the same values, and therefore not different. Finally, another technique will be to replace each XPath with a constant. This action, like in the earlier case, will make the DOM comparison algorithm see no difference in the values of these XPaths. Regardless of the method used to exclude the Delta(X) from the DOM, this process will be transparent to the DOM comparison algorithm. As a result, two new DOMs t_1' and t_2' are produced.

These two new DOMs can now be sent to the comparison algorithm. The state diagram in Figure 3 shows how our proposed algorithm applies to the crawling paradigm in general.

In addition, when computing Delta(X), to further increase the differences between two consecutive loads, the crawler could redirect one of the two requests through a proxy. Practice shows that web pages may display different content based on the origin of the request, for instance users from different countries or even provinces may see different advertisements when visiting the same web page. Thus far, the proposed algorithm keeps track of the Delta(X) and excludes it from the DOM comparison method.

Alternatively, one could keep track of the parts in the DOM that do not change in time and consider only those for the DOM comparison method. Those common parts of the DOM would in this case act like a mask to the current DOM. Regardless of the technique, the effect will be the same (i.e. the Delta(X) will be excluded from the data sent to the DOM comparison function).

A simple experiment of the technique conducted on 30 popular websites (see Appendix for the list) show that only 4 out of 30 (13%) did not change their content on two consecutive loads (the time between the consecutive loads is 10 seconds) [15]. The differences in the 26 other sites proved to be advertisements links, usage statistics, or timestamps that must be ignored by crawlers. Failure to do so may lead to the creation of a large or even infinite number of states, since the state equivalence method will likely separate the states based on these differences.

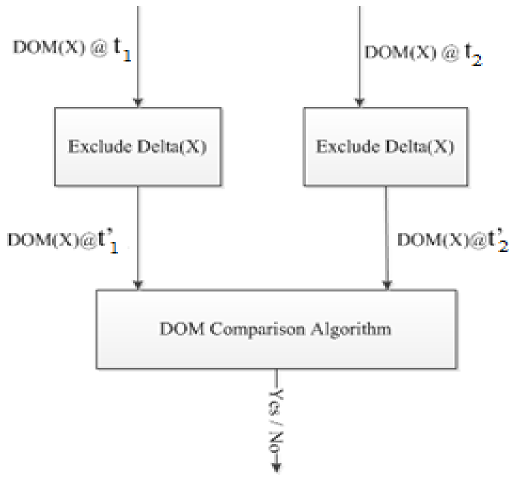


Figure 3. Integration design between a DOM comparison method and the proposed technique

B. Identifying Session Variables and Parameters

Web sites usually track users as they download different pages on the web site. User tracking is useful for identifying user behavior, such as identifying purchasing behavior by tracking the user through various page requests on a shopping-oriented website. Since HTTP is stateless, most modern server-side technologies keep the state information on the server and pass only an identifier between the browser and the server. This is called session tracking. All requests from a browser that contain the same identifier (session id) belong to the same session and the server keeps track of all information associated with the session. A session id can be sent between the server and the browser in one of three ways:

1) As a cookie
2) Embedded as hidden fields in an HTML form
3) Encoded in the URLs in the response body, typically as links to other pages (also known as URL-rewriting)

Among the three methods, using cookies is the most common. Although URL-rewriting is not the most common, many web application servers offer built-in functionality, to allow the application to run with browser clients that do not accept cookies. The method of URL-rewriting works as follows. When the user requests a page of a web site with a URL that does not have a session identifier, a session identifier is created for this user and the user receives a version of the entry web page in which links on the page are annotated by the session identifier. That is, each URL in the response page contains the created session identifier which is usually a string of random characters. When the user selects a link, the web server parses the session identifier from the URL, attaches the same session identifier to the local links on the next generated web page, and returns that web page to the user. The web server continues to parse and attach the session identifiers as long as the user requests a page who’s URL has a session identifier.

As an example of the difficulties caused by not detecting the session identifiers, consider the situation of a web crawler which has found multiple URLs without session identifiers and pointing to pages in the same website. Let's for the sake of simplicity assume that there are two such URLs, called URL_1 and URL_2. A typical crawler will collect URL_1 and URL_2 and put them in its queue to process. If the server hosting the website pointed by URL_1 and URL_2 is using URL-rewriting, when the crawler requests URL_1 the server will notice that URL_1 does not contain a session identifier, so the server will produce a session identifier and will return a page where all URL's contain the generated session identifier. When the crawler requests URL_2, the server again will notice URL_2 does not have a session identifier and generate a session identifier, most probably different from the first one that is produced when URL_1 was requested. Thus two sessions have been created for the crawler. It is very likely that the crawler can find in two different sessions URLs that are pointing to the same page within website. However, since those URLs are found in different sessions, the crawler will not be able to understand that they are actually the same URLs except for different session identifiers. The crawler would thus crawl the same web pages redundantly, thus wasting the crawler's time and bandwidth (and if the crawling purpose was content indexing, filling the search engine's index with duplicate pages, thus wasting storage space).

Another problem faced by automated crawlers, not being able to detect session identifiers is session termination. If the client fails to provide the correct session identifiers, the web application will terminate the session and the crawl operation will result in poor application coverage.

The current solutions for these problems are not reliable. They are based on heuristics, such as known session id name patterns or entropy of the values. There are two sources of weakness with the current solutions: 1) they rely on expert knowledge to create; 2) they depend on common practices that servers use to populate these values. That is, the current solutions require human intervention (not automated) and cannot be effective in case of a server that does not use one of the common practices.

For example in the case of URL-rewriting session identifiers the relevant value will be passed in the request path.

GET /S(120fd4ovfqyogf34f)/home.asp

HTTP/1.1

Alternatively, some web application developers may implement their own version of the mechanism. In order to handle such a case the web crawler has to be preconfigured to identify the session id value. Since such combinations are left to the creativity of web application developers, it is almost impossible to maintain a reliable set of heuristics in order to identify URL-rewriting session identifiers.

Thus, there is a need to effectively identify web sites that contain session identifiers and to be able to identify all the parameters and cookies that need to be tracked by the crawling engine in order to improve web crawling and successfully perform the scan. Unless prior knowledge of this fact is given to a crawler, it would find an essentially unbounded number of URLs to crawl at this one web page alone.

One fundamental point about the session identifiers is that they are by definition unique for a session. This applies to any request component regardless of the place or time when it is constructed. There is also the case where a session id will maintain the same value for subsequent sessions, but will expire after a period of time (session timeout). In this case the algorithm we propose can be reapplied after the session has expired and the session id can be identified this way.

Based on these considerations, we propose a technique which compares all the request components recorded during two different login sequences. As a result of this comparison, any variable that changes its value between these two different login sequences is flagged as a session identifier used by the crawling engine. Conversely, any variable that has the same value in two recorded sequences will not be considered as being part of the session identifiers set.

The proposed algorithm requires that the two recordings of the log-in sequence are done on the same website, using the same user input (e.g. same user name and password) and the same user actions. Failure to respect this requirement will lead to invalid results.

It is also important that the session is invalidated between the two recording actions or that the second recording action will occur when the crawling has reached an out-of-session state.

Let's take for example the following two recorded login sequences. The entities in the square brackets are parameters, passed as part of the POST request to the server such as user name, password etc.

1) Sequence 1
1.1 Request 1

GET https://www.site.com/bank/login.php HTTP/1.1

Cookie: PHPSESSIONID = c9bqcp9w97qgfq4w;

1.2 Request 2

POST https://www.site.com/;GpJLFBYlVprxP8Qky/bank/login.php HTTP/1.1

Cookie: PHPSESSIONID = c9bqcp9w97qgfq4w;

[sessionid]:[qxp9mt3ohyqb4tq3tocJDS] ,[user]:[jsmith],[password]:[Demo1234]

1.3 Request 3

GET https://www.site.com/;GpJLFBYlVprxP8Qky/bank/main.php HTTP/1.1

Cookie: PHPSESSIONID = c9bqcp9w97qgfq4w;

2) Sequence 2
2.1 Request 1

GET https://www.site.com/bank/login.php HTTP/1.1

Cookie: PHPSESSIONID = caksvkOACSCACC00d2kkqbd;

2.2 Request 2

POST
https://www.site.com/;eJeB5rxXGqadZ1p9/bank/login.php
HTTP/1.1

Cookie: PHPSESSIONID = caksvkOACSCACC00d2kkqbd;

[sessionid]:[sdhaovhaohwoefo29020jf9f],[user]:[jsmith],[password]:[Demo1234]

2.3 Request 3

GET
https://www.site.com/;eJeB5rxXGqadZ1p9/bank/main.php
HTTP/1.1

Cookie: PHPSESSIONID = caksvkOACSCACC00d2kkqbd;

Looking at the differences between these two log-in sessions we notice that the value that precedes the /bank path element is dynamically generated. In addition the value of the PHPSESSIONID cookie also changes between the two logins.

Out of all the parameters, we notice that the value of the session-id parameter changes while the username and password remain the same.

In this technique, the following operations are performed after two login sequences have been recorded:

1) Separate all elements of the request. For example path elements will be separated using predefined path delimiters. The same goes for parameter values which will be separated by body delimiters. It is important to identify the delimiters because two values of the same session id might contain a common substring across different logins.
2) Compare the two sequences and identify the elements that change.
3) Construct session identifier entities that can be handled accordingly by the crawler

As discussed above, web sites that use session identifiers may be automatically identified by comparing in-host links to multiple copies of documents from the web sites. Knowing that a particular web site uses session identifiers and identifying the session identifier variables can enhance web-crawling.

III. Crawling Strategies For RIAs

A crawling strategy is an algorithm that decides how the exploration should continue at any point during crawling. As mentioned, for traditional (non-RIA) web applications, crawling strategies are well-studied [6] [7]. But for RIAs, new crawling strategies are required that are also able to perform event-based exploration. In event-based exploration, the crawling strategy decides from which state, which (unexecuted) event should be taken next. Our group has been working on event-based exploration strategies that can efficiently crawl RIAs.

We measure the efficiency of a crawling strategy on a given application by the total number of events and resets executed (reset is the action of bringing the application back to the initial state by loading its URL) used for discovering all the states in the application. That is, a crawling strategy which is able to discover new states using fewer event executions and resets is more efficient, since the event execution and resets are the operations that dominate the crawling time.

The existing tools for crawling RIAs [8] [9] [10] use one of the two basic crawling strategies that are Breadth-First (BF) and Depth-First (DF) search. Although BF and DF are capable of exploring an application completely, they are not very efficient in most of the cases. One reason is that BF and DF are very strict exploration strategies. For example, DF does not explore any other state further unless the most recently explored state is completely explored. Consider the case where we execute an event of the most recently discovered state and we end up at a state that was previously discovered. In this case DF will go back to the most recently discovered state although the current state (or some state that is much closer to the current state than the most recently discovered state) may still have unexplored events. Making such strict decisions definitely increases the number of events executed by any crawler that uses a DF strategy. (Similar inefficiencies occur with BF where the least recently discovered state is given strict exploration priority). In addition, DF and BF crawling does not make any prediction about the application behavior. In fact, it is possible to increase the efficiency of the crawling if accurate predictions can be made about the application behavior. These predictions may be based on the partial but valuable information collected during crawling. For example, by looking at previous executions of an event (in different states), it might be possible to predict how that event is likely to behave in a state where it has not yet been executed. Or predictions can even be made before even crawling begins by observing some general behavioral patterns in the given RIA. Later these patterns can be used as a guide for the crawling strategy. With these motivations, we have been investigating crawling strategies different from BF and DF.

The crawling strategies we have experimented are designed based on a methodology that we call "model-based crawling" [11]. The basic principle in model-based crawling is to come up with some expectation about the behavior of the application. These expectations may be based on behavioral characteristics observed in most RIAs and can be formalized as a "meta-model" which denotes the class of applications that follow the given behavioral characteristics. Once a meta-model is defined, a crawling strategy can be designed that is efficient for crawling any application that follows the chosen meta-model. Of course, we cannot expect that all applications follow the characteristics of the chosen meta-model and we have to make sure that the strategy is capable to crawl any application (including the ones that deviate from the meta-model characteristics). For this reason, in model-based crawling the strategy is designed in an adaptive way. That is, the steps to take if the application being crawled deviates from the meta-model characteristics is also specified as part of the strategy. In summary, a model-based crawling methodology has three steps to design a crawling strategy:

1) Choose a meta-model.

2) Specify a good strategy for crawling any application that follows the meta-model.

3) Specify how to adapt the crawling strategy in case that the application being crawled deviates from the meta-model.

In accordance with our efficiency definition, when specifying the strategy (steps 2-3), we always aim to discover new states using as few resets and event executions as possible based on the expectations provided by the meta-model characteristics.

One of the strategies we have designed using model-based crawling is called Hypercube strategy [12]. As the name suggests, the Hypercube strategy is based on a Hypercube meta-model. The two main characteristics of the Hypercube meta-model is the independence of events (the execution order of the events do not change the state reached) and the expectation that executing an event does not disable or enable other events. The experiments we have conducted using the Hypercube strategy show that it outperforms the BF and DF strategies.

Another strategy we have experimented with is based on a statistical model. In this strategy, we predict the model of the application using Bayesian statistics collected during the crawling rather than starting with an initial expectation. This probability-based strategy tries to predict which events are more likely to result in a new state based on that event's previous execution history (from different states) and tries to take the action that has the maximum likelihood of discovering a new state. Again the initial experimental results for this strategy show that it also outperforms the BF and DF strategies.

We have also been working on distributed crawling of RIAs by using several crawlers running in a cloud environment.

IV. RELATED WORK

Recently there has been research on crawling RIAs, but none of the work proposes a strategy different than BF and DF. In [8], Mesbah, Bozdag, and van Deursen introduced a tool called Crawljax which aims to produce static HTML snapshots of AJAX websites. DF is used as the crawling strategy in Crawljax. In [9] Duda, Frey, Kossman, Matter and Zhou use a BF strategy to crawl AJAX applications. They also propose a caching mechanism to store the results of the JavaScript calls that result in AJAX requests to reduce the communication cost of the crawler. In [10], Amalfitano, Fasolino and Tramontana introduced a tool called CrawlRIA which automatically generates execution traces using a DF strategy and tries to construct the model of the application based on the generated traces.

With regard to state equivalence used in RIA crawlers, [8] only says that they compute a "hash code" for each DOM to compare the current DOM with the already seen ones. But within their strategy, after executing an event they compare if reached state is different from the previous state (where the event has been executed). They use the so-called Levenstein distance which determines the minimum number of operations necessary to convert one string to another. If the distance is above some threshold, then the state reached is regarded as a new state. Since the distance is not an equivalence relation, this method has the possible problem of incorrectly identifying non-equivalent states as equivalent. In [9], it is also mentioned that their state equivalence is based on comparing the hash value of the DOM. This means that equality is used as the equivalence relation and this is too strict. In [10], a state equivalence relation based on comparing the set of HTML elements of two DOMs is considered. According to this method, two states are equivalent if one contains all the HTML elements of the other as a subset. This inclusion is checked based on the indexed path of the elements, event listeners and event handlers of the element. They have also introduced two variations of this relation. In the first variation, DOMs are required to have exactly the same set of HTML elements, in the other variation, only visible elements with registered event listeners are considered and the index requirement for the paths is removed. The weaknesses of this method are that it requires storing each DOM and the cost of comparing the DOMs is high.

Detecting near-duplicate web documents can also be considered a research area related to state equivalence (see [13] for a survey). Near-duplicate web documents are documents that are exactly the same in terms of their main content but differ in small portion of the documents, such as advertisements, timestamps and counters. The main motivations for this research are (a) increasing the quality of content searching and (b) reducing space requirements of search engines. Most of the methods proposed in this area work in batch mode (once documents have already been discovered) which is not suitable for crawling where the decision of whether two states are the same or not should not be a performance bottleneck. Another problem with these methods is that they are mostly based on calculating a similarity measure between two documents; hence they are not equivalence relations in the mathematical sense.

In [14], Bar-Yossef, Keidar and Schonfeld try to address the problem of detecting different URLs that point to similar pages. Given a list of URLs, they first try to find rules to detect URLs that are likely to point to similar documents. Later they try to validate these rules by sampling.

V. CONCLUSION

Crawling is essential for security testing of web applications. RIAs have created new challenges for crawling, and thus security testing of RIAs will not be possible unless these challenges are addressed.

In this paper, we have presented a brief overview of the issues of RIA crawling, with an emphasis on state equivalence. We have proposed two new techniques that can improve the accuracy of state equivalence relations and thus crawler efficiency. We have also presented a summary of our recent work on designing efficient crawling strategies.

ACKNOWLEDGMENTS

This work is supported in part by IBM Canada CAS Research and the Natural Science and Engineering Research Council of Canada.

Disclaimer

The views expressed in this article are the sole responsibility of the authors and do not necessarily reflect those of IBM.

Trademarks

IBM and AppScan are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the Web at “Copyright and trademark information” at www.ibm.com/legal/copytrade.shtml.

References

[1] J. Bau, E. Bursztein, D. Gupta, and J. C. Mitchell, "State of the Art: Automated Black-Box Web Application Vulnerability Testing," in *IEEE Symposium on Security and Privacy*, 2010, pp. 332-345.

[2] Jesse James Garrett. (2005) Adaptive Path. [Online]. http://www.adaptivepath.com/publications/essays/archives/000385.php

[3] World Wide Web Consortium (W3C). (2005) Document Object Model (DOM). [Online]. http://www.w3.org/DOM/

[4] IBM. (2012, Feb.) IBM Rational AppScan Enterprise 8.5 Help Page. [Online]. http://publib.boulder.ibm.com/infocenter/asehelp/v8r5m0/index.jsp?topic=%2Fcom.ibm.ase.help.doc%2Fhelpindex_ase.html

[5] K. Benjamin, G.v Bochmann, G.V Jourdan, and I.V Onut, "Some Modeling Challenges when Testing Rich Internet Applications for Security," in *First International workshop on modeling and detection of vulnerabilities (MDV 2010)*, Paris, 2010.

[6] A. Arasu, J. Cho, A. Garcia-Molina, A Paepcke, and S Raghavan, "Searching the web," *ACM Transactions on Internet Technology*, vol. 1(1), pp. 2-43, 2001.

[7] S. Brin and L. Page, "The Anatomy of a Large-Scale Hypertextual Web Search Engine," *Computer Networks and ISDN Systems.*, vol. 30(1-7), pp. 107-117, 1998.

[8] A. Mesbah, E. Bozdag, and A. Deursen, "Crawling Ajax by Inferring User Interface State Changes," in *Proceedings of the 8th International Conference on Web Engineering, IEEE Computer Society,* 2008, pp. 122-134.

[9] C. Duda, G. Frey, D. Kossmann, R. Matter, and C. Zhou, "AJAX Crawl: Making AJAX Applications Searchable," in *IEEE 25th International Conference on Data Engineering*, 2009, pp. 78-89.

[10] D. Amalfitano, R. Fasolino, and P. Tramontana, "Rich Internet Application Testing Using Execution Trace Data," in *Proceddings of Third International Conference on Software Testing, Verification, and Validation Workshops* , Washington, DC, USA, 2010, pp. 274-283.

[11] Gregor v. Bochmann, Mustafa Emre Dincturk, Guy-Vincent Jourdan, and Iosif-Viorel Onut, "A Model-Based Approach for Crawling Rich Internet Applications," *unpublished*.

[12] Kamara Benjamin, Gregor von Bochmann, Mustafa Emre Dincturk, Guy-Vincent Jourdan, and Iosif Viorel Onut, "A Strategy for Efficient Crawling of Rich Internet Applications," in *Web Engineering: 11th International Conference, ICWE 2011,Paphos, Cyprus*, Sören Auer, Oscar Díaz, and George Papadopoulos, Eds.: Springer Berlin / Heidelberg, 2011, vol. 6757, pp. 74-89.

[13] J. Prasanna Kumar and P. Govindarajulu, "Duplicate and Near Duplicate Documents Detection: A Review ," *European Journal of Scientific Research*, vol. 32, no. 4, pp. 514-527, 2009.

[14] Ziv Bar-Yossef, Idit Keidar, and Uri Schonfeld, "Do not Crawl in the DUST: Cifferent URLs with Similar Text," in *WWW '07: Proceedings of the 16th international conference on World Wide Web*, New York, 2007, pp. 111-120.

[15] Kamara Benjamin. "A Strategy for Efficient Crawling of Rich Internet Applications. EECS-University of Ottawa, Ottawa, Master’s Thesis 2010. [Online]. http://ssrg.site.uottawa.ca/docs/Benjamin-Thesis.pdf

Appendix A: Web Applications for Testing “Load, Reload”

1	http://www.netflix.com
2	http://www.facebook.com
3	http://www.wachovia.com
4	http://www.youtube.com
5	http://www.logicbuy.com
6	http://www.wikipedia.org
7	http://www.amazon.com
8	http://www.ebay.com
9	http://www.live.com
10	http://www.engadget.com
11	http://www.craigslist.org
12	http://www.msn.com
13	http://www.apple.com
14	http://www.bing.com
15	http://www.google.com
16	http://www.foursquare.com
17	http://www.vark.com
18	http://www.ikea.com
19	http://www.www.un.org
20	http://www.gmail.com
21	http://www.godaddy.com
22	http://www.bananarepublic.com
23	http://www.onelook.com
24	http://www.bankofamerica.com
25	http://www.kayak.com
26	http://www.kbb.com
27	http://ssrg.site.uottawa.ca
28	http://www.reuters.com
29	http://www.newegg.com
30	http://www.rapidshare.com