Detecting runtime anomalies in AJAX applications through trace analysis

CS-TR-4989

Jeff Stuckman[*]
University of Maryland, College Park

James Purtilo
University of Maryland, College Park

August 10, 2011

Abstract

AJAX applications are prone to security vulnerabilities due to the ease of inadvertently entrusting the client with security-critical logic. We characterize exploits of such vulnerabilities as violations of a protocol implicitly defined in the client-side code, and we introduce a method to detect and prevent these protocol violations in middleware, without having to modify the original application. We accomplish this by instrumenting the client code to send fragments of execution traces to the server, allowing the server to efficiently prove that the incoming message complies with the protocol. By combining replay execution and constraint solving, our method exploits the componentized structure of applications to minimize the server computing power and network bandwidth required to monitor them. A prototype running on the Google Web Toolkit platform demonstrates our method.

1 Introduction

AJAX applications have become popular for distributing client-server applications to a wide audience, due to the ease of deploying an interactive, client-server application that the user does not need to install. The interactivity of AJAX applications comes at a cost, however, because putting logic on the client that otherwise would have been on the server makes it easier to introduce security vulnerabilities. The increased responsiveness of many AJAX applications comes from moving control flow and presentation logic to the client, making the server a simple provider of services. These applications may appear to be secure when used as intended, but when the client-side code is maliciously modified, or when a user sends malicious traffic directly to the server, the attacker could potentially access or modify unauthorized data, or even compromise the server by instructing it to execute the attacker's code. Aside from the potential security problems, AJAX applications are prone to difficult-to-find bugs because they are vulnerable to changes in the way that web browsers execute code, which can introduce bugs which are dependant on a specific version of a web browser.

Defect-free software is rare, and anomalies that commonly appear in web applications let attackers take advantage of developers' mistakes. [1] classified such attacks as bypass attacks, while [2] called them request integrity attacks and defined them as violations of the developer's intended workflow. Either way, web applications experience such attacks because control flow and navigation logic is inherently visible to the client, especially in more complex AJAX applications. Attacks are performed by tampering with the internal state of the application, forging client requests, or modifying the client-server communication stream. A real-world example of such an attack is when the Oklahoma Department of Corrections website allowed hackers to view tens of thousands of Social Security numbers and other personal information, accomplished by sending a modified request to the server [3]. [4] observed that such vulnerabilities are common, and noted that AJAX applications are especially vulnerable because relocating logic from the server to the client is inherently risky.

[*]I thank the United States Office of Naval Research, which partially supported this research under contract N000140710329.

Whether they are intentional (caused by a malicious user) or unintentional (caused by a web browser incompatibility), we classify these adverse events as *runtime anomalies* occurring on the client side. We define a runtime anomaly as an event where the code has entered a state which should not have been possible, given the sequence of messages exchanged between the client and server so far, while assuming that the browser is executing the code developed for the website. Such anomalies can be detected when the externally observable behavior of the client – that is, the sequence of messages that the client has sent to the server – deviates from what would have been possible in the absence of runtime anomalies. Under this definition, many kinds of exploits, such as buffer overrun and SQL injection attacks, can be classified as runtime anomalies. Previous security research focused on detecting and blocking these exploits in several ways, such as adding security annotations to the application during development to prevent the developer from putting security-critical code on the client [5], requiring the developer to explicitly define a workflow and ensuring that clients comply with it [2], automatically extracting such a workflow using static analysis [6], or recording and analyzing UI events on the client side to detect an anomaly [4].

Detecting anomalies is straightforward if the correct behavior of the client program is defined by a formal specification, but completely specifying the behavior of an application is rare due to the amount of effort required to completely do so. We consider an alternate perspective, where the correct behavior of a client-server application is defined by the code as it was originally deployed. If the behavior exhibited by the client and server is consistent with the code that was deployed, then we deem the behavior to be correct. From the perspective of an observer monitoring network traffic between clients and servers, the client and server programs specify an *implicitly defined protocol*, where the protocol allows any sequence of messages that might be exchanged during a valid run of the client and server programs. We then *verify* a message being sent from the client to the server by checking if it conforms to this protocol. Hence, the behavior of the client passes verification in the absence of observable runtime anomalies.

2 Runtime model checking

One approach to verifying a message from the client is to check if the message is consistent with a model of correct behavior, which was extracted from the client source code or explicitly specified by the developer. An example of this is found in [6], where static analysis on the client-side code produces a pattern of messages that the client may transmit. In this way, the protocol implicitly defined by the client and server code is partially extracted through static analysis. Another approach is found in [2], where system administrators explicitly define a workflow that users should follow. Deviations from this workflow are considered to be "request integrity" violations, similar to our runtime anomalies.

Assuming that the program running on the client is deterministic and runtime anomalies are not occurring, a message sent from the client to the server is a function of the history of messages exchanged between the client and server and the input received from the user so far (where input consists of any interaction the user has had with the UI). Because the message history is known, the server can use constraint solving to check if any combination of user inputs is consistent with the message history. If no such combination of user inputs exists, then a runtime anomaly has been detected. This approach is guaranteed to detect all detectable runtime anomalies, but it quickly becomes intractable for unrestricted applications. The approach is more successful in cases where the extent of the search can be restricted. For example, a system to detect cheating in online games was presented in [7], which was tractable because iterations through the game's event loop could be analyzed independently.

Traces will improve constraint solving efficiency. Although it may be computationally expensive to verify an incoming message, augmenting the contents of the message with extra information could speed verification, if the extra information is carefully chosen. For the same reason, some programs output certification trails (extra information that's not part of the desired output of a program) to improve the performance of result checkers [8]. One obvious way to enhance verification performance with extra information is for the client to send its execution trace along with messages. Execution traces have been used to verify software in the past to specify software in terms of legal traces [9] and to detect probable intrusions through runtime trace monitoring [10].

If the client sent a trace of its entire execution along with each message sent to the server, verifying the client's observable behavior would be easy, because the trace could be checked against the client source code and the message history, allowing the server to efficiently determine that the incoming message is consistent with the client code as authored. An easy optimization is possible by removing entries from the trace when they could be inferred from the source code and the state of the system after the preceding entry. In other words, the client need not record when instructions are executed deterministically. Because the AJAX applications that we are concerned with do not exhibit true nondeterminism, the only nondeterministic steps (from the perspective of the verifier) are those that occur when UI events are triggered by the user, such as when a character is typed into a text box. Such events are sometimes implicitly handled by the browser (such as in the case where a character automatically appears in a text box when the user presses a key), but they can also be hooked and handled directly. We refer to such statements as *trace-generating statements* because modifying the client code to trace those statements (by recording the values that are read and written while executing them) would enable efficient verification.

In the absence of runtime anomalies, a full trace can be efficiently reconstructed from only the trace-generating statements, and the trace will match the observed behavior (the last entry in the trace encodes the generation of the same message that the server received from the client). In the presence of a detectable runtime anomaly, no combination of user inputs is consistent with the message history; therefore, any full trace recovered from the trace of UI interactions would prove to be invalid. Either way, adding trace-generating statements to selected statements enables efficient message verification without affecting the verification result. The Ripley project [4] used a similar concept to detect runtime anomalies in client-server applications built with Microsoft's Volta framework, by recording a transcript of user interface interactions and periodically sending them to the server to facilitate replay execution.

3 A hybrid approach for detecting runtime anomalies

Pure constraint solving and trace analysis are effective for detecting runtime anomalies, but constraint solving is computationally intensive and the bandwidth consumed by traces is potentially high. We define a hybrid approach which combines these two methods while reducing the drawbacks associated with both.

In web applications, events are generally handled by user interface widgets which mediate interaction between application logic and the GUI. Regular AJAX applications use standard HTML controls as their widgets, which provide UI interactivity without having to explicitly trap mouse clicks and keypresses. More complicated frameworks, such as the Google Web Toolkit, allow users to build complex UI widgets that further abstract the HTML DOM from the user. The presence of widgets suggests an alternative approach toward verification – instead of tracing every event when the user interacts with the application, we place trace-generating statements at the *boundary* between the widget the application code, capturing any data returned when the locus of control passes from the widget to the main application. The recorded data includes the return values of function calls into the widget and events that were initiated by the widget but handled by the application. Placing trace-generating statements at the boundary of these two tiers partitions the verification task into two steps. First, the trace is used to verify the client's behavior through "replay execution" (executing the client-side code on the server, while using the trace data as a surrogate for the UI widgets). Next, the trace statements themselves are verified by using a simplified constraint solving process that determines if any valid user actions would produce the output of the widget described in the trace, given a properly functioning widget that received the same inputs that were sent during replay execution.

Verifying the client's behavior through replay execution is simple. Such execution typically begins with an event, specified by the first entry of the trace, that is bound to an entry point of the code. Starting with the state of the client after the previous message was processed, the client code is executed on the server side, substituting widget method calls with simple stubs which return the value that was attached to the current trace entry. When the simulated client code sends a message to the server, the message is compared with what was received, and the verification succeeds if the messages match. The next step – verifying the integrity of the trace – is harder, because the user's ability to interact with the widget introduces nondeterminism. Thus the next step is to use constraint solving to determine if a sequence of unobserved user interactions could place the widget in states that explain the entire history of the widget's output (since its construction). Consider the case of a checkbox that can be enabled and disabled. If the checkbox was enabled when it was constructed, a single call to read the checkbox's value will always validate, because the user had the chance to check or clear the box. A run of calls to read the checkbox's value is valid if the returned value is the same every time. If two successive calls to read the checkbox's value return different values, then the second call only validates if the checkbox was enabled at some moment after the first call was made.

Rules such as the above for checking the constraints of a widget work well for UI widgets built into the browser or a framework such as the Google Web Toolkit, because these rules can be hardcoded into the verifier. When rule-based verification is not possible, widgets can be verified by using a generalized constraint solver, an approach used in [7], which screened for runtime anomalies by using symbolic execution. Combinations of these two constraint solving techniques would be appropriate in cases where more complex widgets are built by composition. For example, in the case where a complex UI widget is authored by composing several HTML DOM objects, the trace validation logic for the DOM objects could be expressed symbolically and supplied to the constraint solving for the general widget. Alternatively, the composed widget could be treated as ordinary code, with its interactions with its component widgets traced (this is the approach we took in the prototype described in the next session). In cases where widgets interact with each other without the interactions being mediated by the main application code, such an approach is mandatory because constraint solving becomes impossible for a widget in isolation.

4 A runtime verifier for the Google Web Toolkit

To test our approach for detecting runtime anomalies by monitoring incoming messages in an AJAX application, we developed an add-on for the Google Web Toolkit [11] framework that monitors running applications and blocks incoming messages when they were the result of a runtime anomaly. The objective is to block some attacks without requiring the developer to explicitly request protection or take any special precautions. This ensure that applications will be protected against unknown attacks that the developer did not anticipate. We sought to develop our prototype as a simple add-on to the GWT which could defend applications without any special effort or skill required on the part of the developer or administrator.

The GWT is a framework which eases the development of AJAX applications, shielding developers from the mechanics of several tasks: transferring data between the client and the server, supporting multiple browsers, and working with HTML objects. Both the client-side and the server-side code is written in Java. The client-side code is developed with the GWT client class libraries, which provide the user interface widget framework and the RPC services. After developing the application, the GWT compiler produces a Javascript file from the client-side code and the client-side class libraries, which is then directly deployed to servers. Meanwhile, the server side code extends a GWT base class, which processes incoming RPC requests, making it directly deployable to a servlet container. Messages sent from the client to the server encapsulate an RPC request, which consists of a call to a specified server method with specified parameters. The server executes the specified method and sends a message back to the client with the result. These are the only kinds of messages exchanged between the client and server.

We developed our verifier to protect GWT applications because the GWT encapsulates user interfaces in a hierarchical widget framework, giving us a natural place to insert trace-generating statements on the widget level. Using the GWT also eases enhancement and monitoring, because of the breadth of framework services that can be modified. This gives us the opportunity to deploy security enhancements without having to rewrite the user's application – having the source code to the application is not even necessary, because the GWT compiler operates on binary class files. We modified the GWT client-side class library to send partial execution traces to the server, and we enhanced the GWT server base class to screen incoming requests for runtime anomalies.

We modified the GWT client-side libraries to collect and buffer partial execution traces in the user's browser and send these traces whenever an RPC request is initiated. The GWT has a large library of UI widgets built in to ease application development. Our prototype traces and data or events passed from built-in widgets to the user's application code. This was accomplished by adding statements to the built-in widgets that record the return values of their methods into a trace buffer whenever an appropriate method is called. Trace statements are only recorded when the return value of the method could not be directly derived without the trace statement. (For example, a call to a button widget asking for its label would not be traced, because the user could not have modified the label since it was set.) The GWT client-side RPC classes were modified to dump the accumulated trace buffer to outgoing messages.

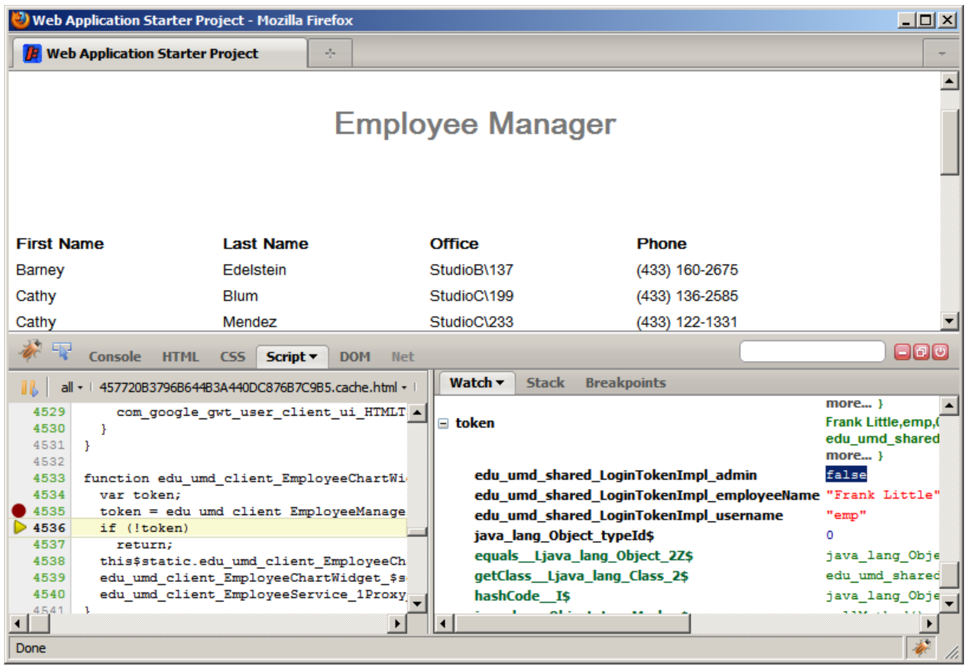
We modified the GWT server classes to parse the traces submitted along with the requests and use them to verify the requests. The system administrator starts the modified server in the same way they start the regular GWT server. When configuring the server, the administrator does need to take the extra step of supplying a JAR file with the client-side code in Java, which is used in the replay execution step.

The server's verification code uses Java sessions to track each simultaneous client of the application. The server maintains one *virtual client machine* for each simultaneous client, which uses replay execution to replicate the state of the client-side code running in the user's browser. Note that, even though Javascript is being executed in the user's browser, the virtual client machines execute the Java code that was compiled into Javascript. This aspect of GWT freed us from having to run a Javascript interpretor in the browser (the Ripley [4] project used the .NET framework to similar effect). When the server receives a message from a client, the server activates the virtual client machine corresponding to the client, and it runs until the virtual client generates an RPC request. If its RPC request matches the request received by the server (with both the method being called and the parameters used to call it matching) without a constraint violation being found in the trace, then the incoming message is valid and it is passed to the server-side service that the client invoked in the first place. Finally, once the server method returns, its return value is copied and passed to the virtual client machine, to ensure that the virtual client observes the same messages that the actual client did. Each virtual client machine runs with its own classloader, to isolate user sessions from each other and to ensure that RPC calls and other calls to the GWT API can be trapped and handled appropriately. To avoid classloader mismatches introduced into the virtual client machine by the server services, RPC parameters and return values are copied and compared with a field-by-field recursive marshaling process (with special handling for Java collections and value types.)

Note that the virtual client machine cannot simply execute the client-side user code, because this code will incorporate GWT UI widgets, for which the trace statements act as surrogates. This means that we needed to substitute widget calls with calls to the trace processor, verifying the consistency of the trace statements themselves in the process. We did this by writing a library of stub classes which mirror the GWT client side classes. The classloader managing the virtual client machine loads the stub classes instead of the GWT client classes, allowing the unmodified client user code to transparently call them. When the client code manipulates the state of the stub classes, the stub classes record this state (sometimes in an abstracted way) to mirror the state of the client's UI widgets. When the stub classes need to return values back to the client code, they read the next value from the trace that was attached to the incoming RPC request message, which is the value that was returned by the corresponding method when it was actually executed by the client. The stub method passes this value to the emulated client code, after first checking if the value is consistent with correct control behavior. This correctness check is done by using constraint-checking rules as described in Section 3. For example, two consecutive calls to get the text from a text box should return the same value, unless the text box was editable and able to receive events at some moment between the two calls. In this way, the consistency of a value being returned from a stub method may depend on values that the stub previously returned, so some stubs must store complex state information which encompasses which values they previous returned, hence dictating which values they may return in the future.

5 A demonstration of our verifier prototype

We demonstrate our prototype by developing an application in the Google Web Toolkit which contains a security vulnerability that our verifier can prevent from being exploited. This application is a simple employee database viewer, where users can log in as either as a superuser or an ordinary user, with superusers being allowed to see private data such as social security numbers. For simplicity and to reduce confounding variables when benchmarking, the employee data is hardcoded into the server side of the application instead of being


Web Application Starter Project - Mozilla Firefox
Web Application Starter Project
Console
HTML
CSS
Script
DOM
Net
Watch
Stack
Breakpoints
all
457720B3796B644B3A440DC876B7C9B5.cache.html
Done

Employee Manager

First Name	Last Name	Office	Phone
Barney	Edelstein	StudioB\137	(433) 160-2675
Cathy	Blum	StudioC\199	(433) 136-2585
Cathy	Mendez	StudioC\233	(433) 122-1331

```
    com_google_gwt_user_client_ui_HTMLT
  }
}

function edu_umd_client_EmployeeChartWi
  var token;
  token = edu umd client EmployeeManage
  if (!token)
    return;
  this$static.edu_umd_client_EmployeeCh
  edu_umd_client_EmployeeChartWidget_$s
  edu_umd_client_EmployeeService_1Proxy
}
```

```
                                                 more... }
token                                            Frank Little,emp,(
                                                 edu_umd_shared
                                                 more... }
    edu_umd_shared_LoginTokenImpl_admin          false
    edu_umd_shared_LoginTokenImpl_employeeName   "Frank Little"
    edu_umd_shared_LoginTokenImpl_username       "emp"
    java_lang_Object_typeId$                     0
    equals__Ljava_lang_Object_2Z$                java_lang_Obje
    getClass__Ljava_lang_Class_2$                edu_umd_sharec
    hashCode__I$                                 java_lang_Obje
```

Figure 1: An example of how the test application can be exploited with the Firebug debugger. The user sets a breakpoint in the GWT generated Javascript (lower left) to modify the login token (lower right).

drawn from a SQL database. The application has a security vulnerability where the user's administrator status is stored in a token on the client side and submitted with each request for data. Due to incorrectly entrusting the client code with security-critical data, ordinary users can view private data by editing their login tokens with a Javascript debugger (Figure 1).

When running the application without our verifier, these hacking attempts succeed. When our prototype is protecting the application, however, any requests for data fail with an error message after the user has modified their login token. This is because manually modifying the login token causes it to mismatch the token stored in the virtual client machine running on the server. Because the login token is submitted as a parameter of the RPC method being called, the virtual client machine will end up with a different method parameter, causing an RPC parameter violation failure. Therefore, this security vulnerability is mitigated by running the test application under our verifier, without taking any special action to detect it, or even modifying the application at all.

6 Performance impact of our approach

We then measured the performance impact associated with using our verifier in this simple scenario. Tests were performed on a closed network with no other traffic, with the server running Google Web Toolkit 2.0 and the Jetty 7.1.1 web server on a ram disk, hosted by the Ubuntu 10.04 LiveCD distribution. The client was running Mozilla Firefox 3.6.6 and the Firebug 1.5.4 debugger on Windows 7 with no other applications running during the test.

6.1 Response time

We first compared the response time of the GWT server with our prototype and our test application with trace-generating statements versus an unmodified GWT server and application. Response time was measured as the time elapsed between the execution of the first and last statements of the GWT request-processing

Table 1: Two response times measured with and without our message verifier prototype installed

	μ (ms)	s (ms)	N
Login time	.75	.55	20
Login time with verifier	20.5	4.35	20
Request time	3.72	1.22	620
Request time with verifier	4.28	1.44	620

Table 2: Sizes of request messages sent from the client to the server during three kinds of requests generated by our test application, with and without the verifier enabled

	Bytes	Bytes with verifier
Login request	190	237
First request	210	233
Subsequent request	211	240

servlet. This time encompasses the time taken when running all user code, including our verification code. The server was driven by the actual client application, with a driver script injected by Firebug that causes rapid requests to be generated. The script logs in as an ordinary user (generating a login request) and then flips back and forth between pages of employee data 30 times, with one flip occurring every .25 seconds (making the request rate independent of the server response time). This process was repeated 20 times for both the modified and unmodified application, along with one warmup run of the process (with its data being discarded). Each time the process was repeated, we started a new session of the application by opening a new browser window, in order to clear the login cookie and expend the time required to create the virtual client machine during each login request.

In Table 1, we show the impact of our verifier on two response times – the time to process the login request (which also includes the time required to build the Java servlet session and the virtual client machine) and the time to process an ordinary request (which only includes the request processing and validation times).

6.2 Javascript download size

Our verifier adds trace-generating and trace-sending code to the GWT client libraries, which are compiled into Javascript and packaged with the client application code. For this reason, the size of the Javascript (which is normally cached by the browser after the first download) will be larger when running our verifier. Using the default compilation options, our test application consumed 75,661 bytes without our verifier and 80,671 bytes with our verifier compiled in.

6.3 RPC size

Because our verifier causes trace information to be packaged with each RPC request, more data will be sent from the client to the server. Note the test application sends three kinds of request messages. The login request is the first request that the client makes, in order to verify the username and password. The client then sends the first request, retrieving the first page of results. A subsequent request is sent whenever the user flips to the next or previous page of results. The number of bytes that the client sent the server for each request is shown in Table 2. Note that the sizes of the responses from the server to the client were not affected and are not shown, because our verifier does not cause any additional information to be sent with RPC responses.

6.4 Discussion and notes

These performance measurements show that our verifier caused a measurable but small performance impact when running our test application, suggesting that this technique could plausibly be effective for real applications. The real-world performance impact will vary depending on the complexity of the application – although more complex applications may generate more network traffic due to calling more UI widgets during each call, the amount of work for the server to handle a baseline RPC request will be larger, reducing the relative performance impact of verifying the message to detect runtime anomalies.

7 Applications and future work

Our prototype demonstrates how the GWT could be modified to prevent attacks in the cases where the attacker had to circumvent the normal user interface of the application to succeed. These modifications also have the benefit of pinpointing some incorrect client behavior caused by browser bugs or transient faults. The widget framework as implemented by the GWT allowed us to keep verification overhead low, because it eliminated the need to simulate an entire DOM in memory. The GWT also gave us an opportunity to effectively instrument the client-side code to generate trace statements by adding the instrumentation to the GWT UI classes, leaving the user code untouched. Some content management frameworks such as Drupal [12] already perform token-based validation to detect when form parameters were maliciously modified, and the Ripley [4] framework implemented an automated system to detect request integrity attacks for applications built with the proprietary, experimental Volta platform. However, to our knowledge, our prototype is the first fully automated tool to monitor an existing application to prevent request integrity attacks while running on an AJAX framework which is publicly available and widely used. Although the best way to ensure security is to carefully develop AJAX applications so the client is not entrusted with sensitive logic in the first place, our defense-in-depth measure could mitigate some attacks that developers miss.

Several practical concerns must be overcome before our verifier could widely be used in production applications. First, we need to expand our library of server-side stub classes and client-side trace-generating code, so constraint checks are performed in all of GWT's built in UI classes. Our prototype only did this for a subset of the UI classes due to the large number of UI classes used in the GWT. Next, the model of user activity must be extended to consider how the user navigates through the application as a whole, rather than considering individual pages in isolation. Our prototype creates a new client virtual machine and clears its client-side state whenever the client navigates to a new page, ignoring the possibility that the client could carry information to the next page in the form of a cookie or URL parameter. Tampering with these cookies or parameters is another avenue for attacking the server, so some mechanism will be needed to ensure that the user is visiting the URLs that the application intended for them to visit, with cookies and parameters that were validly set by previous activity. This is complicated by the fact that users could set bookmarks, allowing them to legitimately begin their session in the middle of a workflow.

Finally, our system must be evaluated for performance with real-world, large-scale applications. If our current prototype proves to generate too much network traffic in a large-scale application, we could modify the way that code is partitioned between constraint solving and replay execution to minimize the extra network traffic generated by a particular application. However, this would cause some user code to be verified by constraint solving, which introduces the need for generalized Java constraint solvers which may not yet be appropriate for general, operational use.

References

[1] J. Offutt, Y. Wu, X. Du, and H. Huang, "Bypass testing of web applications," *Software Reliability Engineering, International Symposium on*, vol. 0, pp. 187–197, 2004.

[2] K. Jayaraman, G. Lewandowski, P. Talaga, and S. Chapin, "Enforcing request integrity in web applications," in *Data and Applications Security and Privacy XXIV* (S. Foresti and S. Jajodia, eds.), vol. 6166 of *Lecture Notes in Computer Science*, pp. 225–240, Springer Berlin / Heidelberg, 2010.

[3] "Oklahoma leaks tens of thousands of social security numbers, other sensitive data.." http://thedailywtf.com/Articles/Oklahoma-Leaks-Tens-of-Thousands-of-Social-Security-Numbers,-Other-Sensitive-Data.aspx, 2008. The Daily WTF.

[4] K. Vikram, A. Prateek, and B. Livshits, "Ripley: automatically securing web 2.0 applications through replicated execution," in *CCS '09: Proceedings of the 16th ACM conference on Computer and communications security*, (New York, NY, USA), pp. 173–186, ACM, 2009.

[5] S. Chong, J. Liu, A. C. Myers, X. Qi, K. Vikram, L. Zheng, and X. Zheng, "Building secure web applications with automatic partitioning," *Commun. ACM*, vol. 52, no. 2, pp. 79–87, 2009.

[6] A. Guha, S. Krishnamurthi, and T. Jim, "Using static analysis for ajax intrusion detection," in *WWW '09: Proceedings of the 18th international conference on World wide web*, (New York, NY, USA), pp. 561–570, ACM, 2009.

[7] D. Bethea, R. A. Cochran, and M. K. Reiter, "Server-side verification of client behavior in online games," in *Proceedings of the 17th ISOC Network and Distributed System Security Symposium*, pp. 21–36, Internet Society, 2010.

[8] D. S. Wilson, G. F. Sullivan, and G. M. Masson, "Certification of computational results," *IEEE Transactions on Computers*, vol. 44, pp. 833–847, 1995.

[9] J. McLean, "Proving noninterference and functional correctness using traces," *JOURNAL OF COMPUTER SECURITY*, vol. 1, pp. 37–58, 1992.

[10] S. A. Hofmeyr, S. Forrest, and A. Somayaji, "Intrusion detection using sequences of system calls," *Journal of Computer Security*, vol. 6, pp. 151–180, January 1998.

[11] "Google web toolkit." `http://code.google.com/webtoolkit/`.

[12] "Form generation – drupal api." `http://api.drupal.org/api/group/form_api/6`.