An Execution-flow Based Method for Detecting Cross-Site Scripting Attacks

Qianjie Zhang, Hao Chen, Jianhua Sun
Adv. Internet and Media Lab
School of Computer and Communication, Hunan University
Changsha, China
E-mail: jhsun@aimlab.org

***Abstract*—We present an execution-flow analysis for JavaScript programs running in a web browser to prevent Cross-site Scripting (XSS) attacks. We construct finite-state automata (FSA) to model the client-side behavior of Ajax applications under normal execution. Our system is deployed in proxy mode. The proxy analyzes the execution flow of client-side JavaScript before the requested web pages arrive at the browser to prevent potentially malicious scripts, which do not conform to the FSA. We evaluate our technique against several real-world applications and the result shows that it protects against a variety of XSS attacks and has an acceptable performance overhead.**

Keywords-XSS; FSA; JavaScript; Ajax

I. INTRODUCTION

JavaScript is the cornerstone of Ajax applications. Ajax developers leverage it to enhance user experiences such as richer user interfaces and lower latency of interaction. But unfortunately, it also increases the possibility of being affected by XSS attacks. It is the most common use of JavaScript to compose malicious codes. XSS vulnerabilities make it possible for an attacker to inject malicious content into web pages generated by trusted web servers. Since the malicious scripts run with the same privileges as the trusted script, they can steal a victim user's private data or take unauthorized actions without users' permission. How to distinguish authorized from unauthorized scripts becomes the key to detecting XSS attacks.

In this paper, we built an execution-flow analyzer for the client side Ajax web applications. We analyze the JavaScript code on the client side and produces FSAs to model client side program behavior. We put these FSAs in a proxy to monitor all execution flow of the browser. If the flow doesn't match the pre-built FSAs, it would be an XSS attack. We consider the following four aspects to design our system:

- JavaScript is the client-side programming language for Ajax applications. It also the most common language for attackers to inject malicious scripts into Ajax. And the injected XSS will change the execution flow against that of the normal JavaScript.
- JavaScript is an interpretive programming language, the web browser must interpret scripts line by line before one web page is loaded. When a browser renders a web page, it parses and executes script codes.
- The web application developers know exactly which scripts should be executed for the application, so other scripts occur at runtime may be potential attacks.
- XSS is usually inserted into the most frequently visited part of the web application. But every coin has two sides, this not only allows an attacker to easily attack the users, but also makes malicious scripts easy to be detected.

Our work is inspired by that of R. Sekar's FSA [2], who used program analysis methods to build system call monitor and intrusion detector for traditional applications. Accordingly, we analyze function call of JavaScript to detect XSS for web applications. The FSA-algorithm captures program behaviors in terms of sequences of function calls of JavaScript. Figure 1 illustrates a sample program and its automaton learned by FSA-algorithm. The automaton includes a set of state nodes S (S = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11}), and a set of edges (E = {f0, f1, f2, f3, f4, f5}). Both S and E are finite sets. Each element in S records a state of the program. If a particular function call in a particular state triggers a transition from that state to another one, that transition is labeled with that function call.

An FSA can capture infinite numbers of sequences of random length using finite storage. Its states and edges can record short and long range correlations. Also, an FSA can be traversed in different ways. So it can capture structures in programs such as loops and branches. Besides these advantages, previous researches on finite-state-based learning have some identified negatives: There is no algorithm for constructing FSAs from function call traces. Instead, they construct FSA states and edges from sequences relying on human insight and intuition. Moreover, how to learning compact FSA is a hard problem. Reference [1] shows that learning perfect FSA is as hard as integer factorization.

Against these drawbacks, we present an automatic and effective method of learning compact FSAs to characterize Ajax behaviors. The main contributions of this paper are summarized as follows: We introduce a solution for mitigating XSS attacks by matching execution-flow of Ajax applications against the pre-build FSAs, which captures the well-behaved client-side execution of JavaScript. Our technology neither requires to modify source codes of Ajax applications, nor be deployed in user's browser. Thus, each web site can be protected from XSS exploits transparently

```
1.  f0;
2.  for (f3) {
3.        f5(f1);
4.        if (f2) …;
5.        else   f3;
6.        f1;
7.        if (…)   f4;
8.        else   f3;
9.        f2;
10. }
11. f4;
```

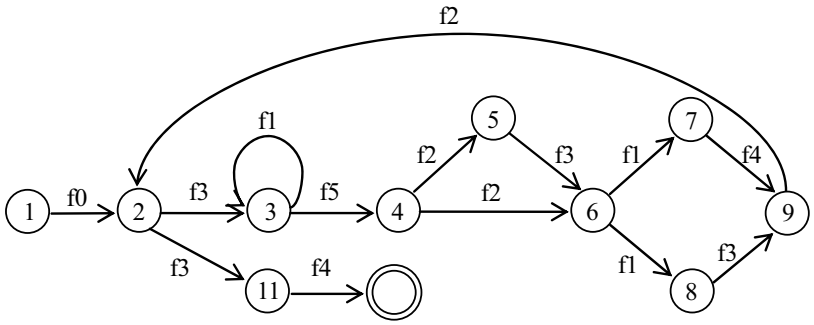

Figure 1. A sample program and its automation learned by FSA-algorithm.

for its visitors. Our system is deployed in a proxy and doesn't need the knowledge of the server-side program. So it can be used with a variety of server technologies. Finally, we demonstrate that our techniques can analyze Ajax's behavior and preventing authentic XSS attacks.

The rest of this paper is structured as follows. In Section Ⅱ we describe how to learn FSAs from function-call sequences of JavaScript. In Section Ⅲ, we introduce our XSS detection system. Section Ⅳ presents the evaluation of our technology against several Ajax applications. In Section Ⅴ, we present related work on detecting and preventing XSS attacks. Finally, Section Ⅵ concludes.

II. LEANING FINITE-STATE AUTOMATA

Our learning algorithm is to trace the normal execution of JavaScript application. As shown of a simplified example in Figure 1, we construct the FSA with function call name, file name and the line number where the call is invoked. Each distinct line number or file name corresponds to a different state of the FSA. Each transition between states is labeled with a particular function call. To construct the transitions, we use both the current triple of *[(File，Line), Func]* and the next triple of *[(NextFile，NextLine), NextFunc]*. The invocation of the current function results in the addition of a transition from the state *(File，Line)* to *(NextFile，NextLine)* which is labeled with *Func*. The construction process continues through many different executions of program, with each time possibly adding more states and/or transitions. Figure 2 illustrates this process.

A. Function Call Tracing

Currently, there are already several tools for function call tracing. For instance, Firebug [3] and LiveHTTPHeaders [4] for Firefox, Web Development Helper [5] and Internet Explorer Developer Toolbar [6] for IE. Most of them are program debugging tools, which analysis a program by But these debuggers aim to test and debug target programs, not suitable for collecting information continuously.

Thus, we chose another dynamic tracing tool DTrace. It is a dynamic troubleshooting and analysis tool first introduced in the Solaris 10 and OpenSolaris operating systems. Tracing programs are written in the D programming language. D scripts consist of a list of one or more probes, and each probe is associated with an action. A probe may analyze the run-time situation by accessing the call stack and context variables and evaluating expressions. Then some information can be printed out or logged. We only use probes related to JavaScript to gather useful information of function-call sequences continuously.

B. Dealing with Event-driven JavaScript

Browsers execute JavaScript in an event-driven fashion. Once JavaScript code of a web page is loaded, users can interact with the page in a variety of ways. As events occur, the browser executes a corresponding event handler to respond to the user interactions. How handlers execute depends on user behavior and the interactions with browser. If we construct only one FSA to model the behavior of an Ajax application, it will be inaccurate. So, we build different FSAs for each JavaScript event to make the analysis more accurate. This also can reduce the number of sequences in real-time monitoring. Since each user interaction only involves several events, not all of the events of a program.

C. Removing JavaScript of Browser

For our system, Dtrace has more obvious advantages than other tracking tools. However, Dtrace works at the operating system level, so it can't distinguish where the current running JavaScript function comes from: the web application or the browser. These events, such as mouse clicks and movements, keyboard presses and the creation and initialization of the JavaScript events, etc. can trigger browser's JavaScript functions. There are 423 script files containing JavaScript in the Firefox 3.0, which contains tens

· [(f1, 1), fun1] [(f1, 3), fun2] [(f1, 6), fun1]
· [(f1, 1), fun1] [(f1, 5), fun3] [(f1, 6), fun1]

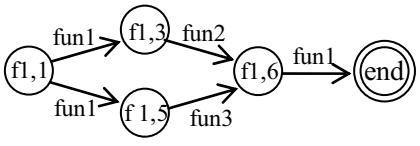

Figure 2. Two traces produced by a program and the generated automaton.

of thousands of JavaScript functions. Thus, in order to construct FSAs only characterizing the Ajax application's behavior, we need to remove the browser's JavaScript from the collected function-call sequences, and satisfy the requirement of high efficiency for real-time monitoring. There are two following difficulties:

- How to distinguish where the current function call is made from.
- How to quickly remove the browser scripts to speed up the FSAs construction and the XSS detection, which have to deal with numerous JavaScript function calls. Especially in the real-time monitoring, because the user is waiting for the response.

For the first issue, since the browser files that include JavaScript codes are known. It is easy to determine where a function call is invoked from. For the second one, we group the browser files by their initials. When a function is invoked, we gather the file name from where the function is called, and then match it against the browser file names which have the same initials. Comparing with the non-grouping method, which needs to match 423 file names each time, the grouping one needs only 16.27 on average.

D. Constructing perfect FSA

An open problem in dynamic analysis is code coverage, which describes the degree to which the source code of a program has been tested. Analysis in [[7]] shows that the degree to the detected vulnerabilities is in direct proportion to the code coverage. For our analysis, due to the factors such as the conditional branches of the program and the event-driven nature of JavaScript, we can't achieve 100% code coverage. If some areas of codes are not been covered, but the real-time detection reach the uncovered areas, it will cause a false alarm. Thus, how to collect the function-call sequences as comprehensive as possible to achieve high code coverage becomes particularly important.

To solve the above problem, we use both automatic and manual methods. To collect the function-call sequences, our analysis needs to simulate users' interactions with Ajax applications. At present there are some existing tools, such as Crawljax [8] and AjaxTracer [9]. They obtain the JavaScript events on a web page by statically analyzing the original code of the page, and then trigger them. Comparing to the manually–trigger method, these techniques help us to gather function-call sequences more comprehensively and automatically. But these automatic tools also have their own disadvantages: they can't simulate users' input well, which is an important part of the users' interactions with Ajax applications. So we also need to simulate the user's input manually. We test the Ajax applications multiple times with multi-cases to achieve high code coverage.

III. XSS INTRUSION DETECTION

A. Runtime Monitoring

We deployed a proxy to protect users from XSS attacks. Figure 3 presents how it works. When a user request is

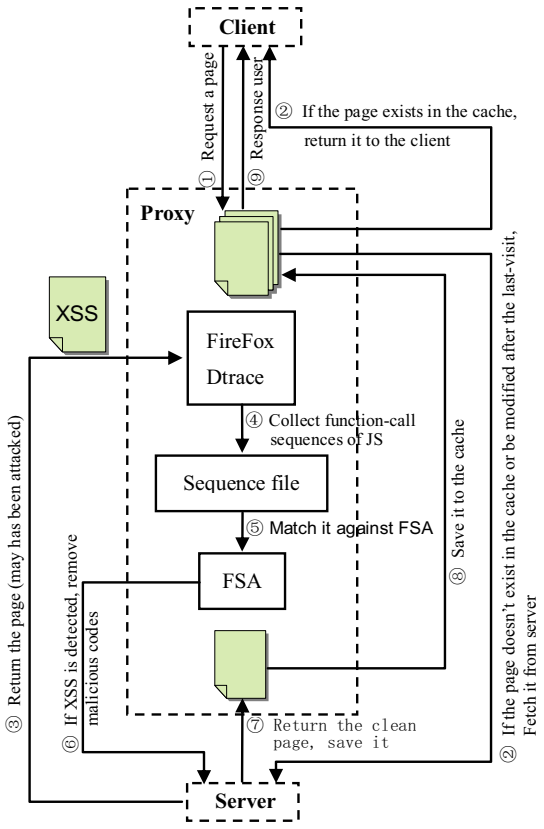

Figure 3. The process of runtime monitoring.

detected by the proxy, it checks whether the requested page exists in the cache. If so, the proxy returns the page to the user. Otherwise, it means the page has not been accessed or modified after the last-visit. Then the proxy fetches it from the server. When the code of the requested page arrives, the proxy executes them in the browser and uses DTrace to collect function-call sequences. The collected sequences are compared with the pre-built FSA. Once an inconsistency is found, it is marked as a potential attack.

The monitor ensures that the function-call sequences match against a sequence of the FSA. For each function call intercepted, we proceed as follows:

- In a given state, the monitor checks if the next function call matches any one of the labels of the out-degree transitions. If so, the FSA follows it to the target state of the transition. If not, mark it as a possible XSS attack.
- Update the state of the FSA to the new state. If the new state is not in the FSA, it is marked as a possible XSS attack.
- For all the possible attacks, to determine whether they are XSS, we need do some further analysis, which is discussed later in the following paragraph in detail. If an XSS attack is detected, we log it and check whether it exists in possible location (such as the back-end database of the Ajax application, XML

documents, etc.). If it exists, it is flagged as an XSS attack. Then we remove the malicious scripts and send the harmless page to users.

1) Reducing the false alarm: During the function-call sequences collection progress, we maximize the code coverage to reduce the false alarm. But it's almost impossible to achieve 100% code coverage due to the growing size of code and complexities of client-side JavaScript. Through an investigation of XSS and its attack modes, we found that victims suffering from XSS attacks are always compromised by malicious content injected into web pages with vulnerabilities, not by modifying the source codes. Thus, when suspicious scripts are detected, we further check where these scripts come from. If they come from the source codes of the program, they are regarded as normal program behaviors and we add them to FSAs to make the training set more completely. Otherwise, it is an XSS.

2) Proxy: Building a proxy based architecture has the following advantages. First, it separates the server from the XSS-prevention system, which does not have the limitation of tightly binding the server being protected to a specific operating system (in our case Solaris). Second, a proxy server accelerates service requests by retrieving content saved for a previous request made by the same client or even other clients. For an attacked page, we remove malicious scripts at the first time when the page is accessed. Then the harmless page is cached. Our proxy could response to users directly when it is accessed at the second time.

B. Post-Intrusion Analysis

Our real-time monitoring has the capability to detect persistent and reflected XSS. Both of their payloads arrive to the server. But there is another kind of XSS called DOM-based XSS which does not rely on the payload embedded by the server in same response page. DOM-based XSS uses some DOM object operations of normal Ajax applications to attack user, entirely on client side. Thus, it can escape from the real-time monitoring.

To detect DOM-based XSS, we present a post-intrusion detection method -- analyzing the attacked web application and constructing FSAs to model its behavior. The process of collecting function-call sequences and constructing FSAs for the attacked program ('Dirty' FSA, DFSA) is exactly the same as the normal program ('Clean' FSA, CFSA). Our post-intrusion analysis detects intrusion by comparing DFSAs to CFSAs: if we detect a new state in DFSA which is not in CFSA, or a transition from current state to the next state in DFSA is not labeled with the same function call as the CFSA, an XSS attack is to be marked. As the post-intrusion analysis is a full function check, regardless of a link that contains malicious scripts or a link navigating to a malicious site to perform an XSS or load Trojan scripts, we can detect the execution of its malicious functions.

IV. Experimental Evaluation

To evaluate the effectiveness of our FSA-based algorithm and techniques, we apply them in several Ajax applications.

- Blog, a blog application which is written by students. It has simple functions, accordingly the number of FSA and its state number are small.
- Jibberbook, a message board application which leverages an external library to filter message content.
- Metatron, a page-based online chat program, which is used as the manager of Project Voodoo.
- AjaxIM, a widely-used online chat application.

A. Summary

Our analysis constructed FSAs for the above applications successfully. We attacked these applications by the most representative XSS, such as XSS listed in *XSS Cheat Sheet* [10], which includes complex examples of XSS attack strings. Many of these examples are capable of damaging real-word web applications and their defenses. To fully exercise vulnerabilities, we embedded attack strings in the pages which were tested. We then used our XSS-intrusion detection system to defend against attacks and evaluated whether our XSS-intrusion detection system worked well. We recorded the number of states of FSAs, the time overhead of the real-time and post-intrusion detection, and the XSS defense effectiveness.

B. XSS Defense Effectiveness

Experiment results are summarized in TABLE I. The living document XSS Cheat Sheet contained 111 vectors at the time of testing, which has 93 XSS attack examples, include 75 persistent XSS, 11 reflective XSS and 7 DOM-based XSS. We embedded all the attack strings into the tested Ajax applications, such as the message board of Jibberbook, the user property and chat window of AjaxIM, which contain XSS vulnerabilities. To load an XSS-based Trojan, we made a little modification to five strings of persistent XSS strings, and injected them into Jibberbook.

For reflective XSS vulnerabilities, they occur commonly in HTTP query parameters or in HTML form submissions, and are used immediately by the server to generate a page for the user. For persistent XSS vulnerabilities, they are stored in the server, and then included in normal pages returned to other users without proper HTML sanitization. In order to attack clients, both reflective and persistent XSS must go through the server. So they can be detected by our intrusion-prevention proxy. As shown in TABLE I, all of the 75 persistent XSS and 11 reflective XSS are detected by both runtime monitor and post-intrusion analysis. For the DOM-based XSS, they attack victims entirely on the client side, which prevents our runtime monitor from stopping them. But we can detect them by using the post-intrusion analysis. We disclosed all the 7 ones shown in TABLE I.

TABLE I. XSS DEFENSE EFFECTIVENESS TEST RESULTS

Types of Attack	The Number of Attack	Runtime Detection	Post-Intrusion Detection
Reflective XSS	11	11	11
Persistent XSS	75	75	75
DOM-Based XSS	7	0	7
Total	93	86	93

C. Time Overhead

To evaluate the overhead of our experiment, we collect some meaningful information which may impact the results of XSS detection. For instance, the amount of code of the Ajax application being tested, it directly affects the state number of the Clean FSAs. And the number of states influences the time overhead of runtime and post-intrusion detection. We list all of them in TABLE II.

Time overhead results for this experiment are summarized in TABLE III. These data are relevant to the tested Ajax applications and the experiment environment. S_{clean} and T_{CFSA} are directly affected by LOC_{js}. Since it is obvious that larger code base of client-side procedures leads to a bigger S_{clean}. In addition, they are related to the writing expression of the program. For instance, in order to prevent attacks, developers of Metatron wrote more than one statement in one line that decreases the readability of the source codes. As S_{clean} related to the lines of code, it is only 58 in Metatron, which is less than other Ajax applications. S_{dirty} depends on the malicious codes which are embedded in the program. For instance, Trojan loads one or more script files into the client-side browser, which makes the state number of victim program's DFSA larger. That is why there is a total of 1781 status of Jibberbook's DFSA (we injected Trojan in Jibberbook), which is more than other Ajax applications and cost more time to detect them.

TCFSA and TDFSA are the time overhead of the construction of CFSA and DFSA, which consists of two parts: T_{trace} and T_{FSA}. T_{trace} refers to the time overhead of gathering the function-call sequences, including the time spent on automated collection and manual collection. T_{FSA} is the time spent on constructing FSA from function-call sequences. They are relevant to the functions of the tested program, the efficiency of the automated tools, etc. T_{CFSA}, T_{DFSA}, T_{first}, T_{second} and $T_{post\text{-}intrusion}$, these time-related information are influenced by many factors, such as the hardware of the proxy server, congestion status of networks, as well as the database. For instance, if the database of Jibberbook stores huge amounts of messages from users, it consumes a lot of time to load them. T_{second} is the time spent on the second-time visit. Since when a page is visited at the second time, the harmless page has been cached by proxy. If the page has not been modified after the first-time visit, the proxy can send the cached content to user directly. So in general, T_{first} is bigger than T_{second}. As seen from the above results, our method can prevent a variety of XSS attacks and has an acceptable performance overhead.

TABLE II. THE INFORMATION COLLECTED IN OUR EXPERIMENT

LOC_{total}	The total code amount of the Ajax application
LOC_{js}	The code amount of client-side JavaScript
S_{clean}/S_{dirty}	The total number of states of CFSA/DFSA
T_{CFSA}/T_{DFSA}	The time spent on CFSA/DFSA construction
T_{first}	The average time spent on visiting every page of the program at the first time
T_{second}	The average time consumption of the second-time visiting
$T_{post\text{-}intrusion}$	The time spent on XSS-intrusion detection after the program is attacked

V. RELATED WORK

By now there have been a variety of defensive techniques to prevent XSS, including the following aspects: static analysis, dynamic analysis, black-box testing, white-box testing, anomaly detection, etc. Generally, these approaches are deployed on the client-side or server-side to protect web users from XSS injection attack.

Server-side protection: At present, many automated testing tools have been in existence, such as black-box testing tools [11] and white-box testing tools [[13], [14]], which are deployed on the server side. They have already been successfully applied in practice. They can help web site developers and administrators to detect the potential XSS vulnerabilities. But the limitation is the significant number of false positives and false alarms. These are all third-part tools, which need the help of web site developers to fix the detected vulnerabilities. This incurs extra costs.

Client-side protection: To remedy the shortcomings of server- side protection, there have been several defensive strategies which are deployed on the client side. In [15], a client-side mechanism for detecting malicious JavaScript is proposed. The system consists of a browser-embedded script auditing component, and an IDS that processes the audit logs and compares them to signatures of known malicious behavior or attacks. With this system, it is possible to detect various kinds of malicious scripts, not only XSS attacks. However, the system has significant weakness: it can only detect the XSS attacks whose behavior haven been known. Attacks that do not anticipated by the signature authors are left unprotected by the scheme.

The two main aims of XSS attacks are stealing the victim user's sensitive information and invoking malicious acts on the user's behalf. Noxes [16] provides a client-side web proxy to block URL requests by malicious content using manual and automatic rules. Reference [17] presents another approach: tracking the flow of sensitive information in the browser to prevent malicious content from leaking such information. Both of these projects focus on ensuring confidentiality of sensitive data (e.g., cookies) by analyzing the flow of data through the browser, rather than preventing unauthorized script execution. They can defeat only the first goal of XSS attacks. It would be defeated by attacks that do not violate same-origin policies. By contrast, our approach is based on analyzing function-call sequences of JavaScript to detect unauthorized scripts, we can defeat both objectives of XSS attacks.

All client-side solutions share the same drawback: the

TABLE III. LINES OF CODE OF APPLICATIONS AND TIME OVERHEAD OF XSS-INTRUSION DETECTION

Ajax App	LOCtotal	LOCjs	Sclean	Sdirty	TCFSA（m）	TDFSA（m）	Tfirst（ms）	Tsecond（ms）	Tpost-intrusion (s)
Blog	1K	300	63	107	5	10	1364	358	14.7
Metatron	2K	1054	58	245	14	18	758	216	15.2
Jibberbook	21K	560	110	1781	11	17	2487	521	30.6
AjaxIM	9K	9725	986	1568	27	34	856	326	45.2

necessity to install updates or additional components on each user's workstation. While most novice users lack of safety awareness, we cannot expect them to deploy a client-side XSS defense, which makes client-side defense be severely limited in practice.

Server and client joint defense: There is an alternative approach in which server and client work in collaboration with each other. The server annotates the delivered content and provides information on the legitimacy or level of privileges of scripts. The Web browser is then responsible for checking and enforcing these annotations.

If web browsers are capable of distinguishing authorized from unauthorized, it can build up a robust XSS prevention system. This vision was first espoused in BEEP [18]. The authors present two policies. First, labeling elements in the HTML source, which are assumed to contain malicious code. So the browser can know whether a script in the DOM tree contains user-provided content or not. The modified browser verifies each script with respect to the policy and denies unauthorized script execution. Second, a whitelist policy, which allows a script to execute only if it is known-good.

These kinds of techniques in which server and client collaborate with each other can better distinguish malicious scripts from normal. But they all share two disadvantages: they need to modify not only the source code of the web application, but also of the browser. However, an ideal XSS defense approach should has no necessary to modify the source code of web applications and install additional components on user's workstation.

VI. CONCLUSION

We present the design and implement of an execution-flow analysis based system for JavaScript programs running in a web browser to prevent XSS attacks. It can produce function-call sequences of Ajax application, so as to build up FSAs to express the normal program behavior. Our system can be deployed in proxy mode. The proxy analyzes the execution flow of client-side JavaScript before the requested web pages arrive at the browser. In addition to the dynamical tracing and realtime protection, the system also supports postmortem analysis of XSS attacks. In general, the system successfully prohibits and removes a variety of XSS attacks, maximizing the protection of web applications.

ACKNOWLEDGMENT

This paper is supported by the National Natural Science Foundation of China under grants 60803130 and 60703096, and the National Key Fundamental Research Program (the 973 Program) of China under grant 2007CB310900.

REFERENCES

[1] M. Kearns and L. Valiant, "Cryptographic limitations on learning boolean formulae and finite automata," J. ACM, vol 41, no. 1, pp. 67-95, 1994.

[2] R. Sekar, M. Bendre, D. Dhurjati, and P. Bollineni. "A fast automaton-based method for detecting anomalous program behaviors," IEEE Symposium on Security and Privacy, 2001, pp. 144-155.

[3] Mozilla. Firebug. http://getfirebug.com/.

[4] Mozdev. LiveHTTPHeaders. http://livehttpheaders.mozdev.org/.

[5] Nikhil Kothari. Web Development Helper. http://projects.nikhilk.net/WebDevHelper/.

[6] Microsoft Corporation. Internet Explorer Developer Toolbar. http://www.microsoft.com/downloadS/details.aspx?familyid=E59C3964-672D-4511-BB3E-2D5E1DB91038&displaylang=en.

[7] M. H. Chen, M. R. Lyu, and E. Wong. "An empirical study of the correlation between code coverage and reliability estimation," IEEE METRICS'96, New York: Albany, March 1996, pp. 133–141.

[8] A. Mesbah, E. Bozdag, and A. van Deursen. "Crawling Ajax by inferring user interface state changes," the 8th Int. Conf. on Web Engineering (ICWE), IEEE Computer Society, 2008, pp. 122–134.

[9] Myungjin Lee, Sumeet Singh, and Ramana Rao Kompella. AjaxTracker. http://www.cs.purdue.edu/synlab/ajaxtracker/.

[10] R. Hansen, "XSS (cross site scripting) cheat sheet esp: for filter evasion," 2008. [Online]. http://ha.ckers.org/xss.html.

[11] M. V. Gundy and H. Chen. "Noncespaces: using randomization to enforce information flow tracking and thwart crosssite scripting attacks," the 16th Annual Network and Distributed System Security Symposium (NDSS), San Diego, CA, USA, February, 2009, pp. 56-64.

[12] Sean McAllister, Engin Kirda, and Christopher Krügel. "Expanding human interactions for in-depth testing of Web applications," the 11th Symposium on Recent Advances in Intrusion Detection (RAID), Boston, USA, September 2008.

[13] Y.-W. Huang, F. Yu, C. Hang, C.-H. Tsai, D. Lee, and S.-Y. Kuo. "Securing web application code by static analysis and runtime protection," the International World Wide Web Conference (WWW'04), May 2004, pp. 40–52.

[14] N. Jovanovic, C. Kruegel, and E. Kirda. "Pixy: a static analysis tool for detecting web application vulnerabilities," the IEEE Symposium on Security and Privacy, May 2006, pp. 258–263.

[15] O. Hallaraker and G. Vigna. "Detecting malicious JavaScript code in Mozilla," IEEE Int. Conf. on Engineering of Complex Computer Systems (ICECCS), Santa Barbara, CA, USA, June 2005, pp. 85-94.

[16] E. Kirda, C. Kruegel, G. Vigna, and N. Jovanovic, "Noxes: a client-side solution for mitigating cross-site scripting attacks," the 21st Annual ACM Symposium on Applied Computing, Dijon, France, April 2006, pp. 330-337.

[17] P. Vogt, F. Nentwich, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna, "Cross-site scripting prevention with dynamic data tainting and static analysis," the 14th Annual Network & Distributed System Security Symposium, San Diego, CA, USA, February 2007, pp. 74-83.

[18] T. Jim and N. Swamy and M. Hicks. "BEEP: browser- enforced embedded policies," the 16th International World Wide Web Conference (WWW2007), Banff, 2007, pp. 601-610.