Arab J Sci Eng (2016) 41:897–920
DOI 10.1007/s13369-015-1891-7

RESEARCH ARTICLE - COMPUTER ENGINEERING AND COMPUTER SCIENCE

XSS-SAFE: A Server-Side Approach to Detect and Mitigate Cross-Site Scripting (XSS) Attacks in JavaScript Code

Shashank Gupta[1] · **B. B. Gupta**[1]

Received: 8 December 2014 / Accepted: 6 October 2015 / Published online: 30 October 2015

Abstract Nowadays, Web applications are considered to be one of the most ubiquitous platforms for providing the information and service release over the World Wide Web, particularly those deployed in health care, banking, e-commerce operations, etc. Boom of social networking sites and modern Web applications that transfer dynamic information to the client-side Web browsers has increased the user-generated and feature-rich HTML content on the Internet. This enhanced HTML content includes a malicious attack vector for Web-related attacks. Cross-site scripting (XSS) attacks are presently the most exploited security problems in modern Web applications and activated by an attacker to utilize the vulnerabilities of the poorly written Web application source code. Users across all over the popular social networking Web sites are exposed to XSS attacks. These attacks are generally caused by the malicious scripts, which do not validate the user-injected input appropriately and exploit the vulnerabilities in the source code of the Web applications. It results in the loss of confidential information such as stealing of cookies, theft of passwords, and other private credentials. In this paper, we propose a robust framework known as XSS-SAFE (Cross-Site Scripting Secure Web Application FramEwork), which is a server-side automated framework for the detection and mitigation of XSS attacks. XSS-SAFE is designed based on the idea of injecting the features of JavaScript and introduced an idea of injecting the sanitization routines in the source code of JavaScript to detect and mitigate the malicious injected XSS attack vectors. We repeatedly inject the feature content, generate rules, and insert sanitization routines for the discovery of XSS attacks. We have evaluated our approach on five real-world JavaServer Pages (JSP) programs. The results indicate that XSS-SAFE detects and mitigates most of the previously known and unknown XSS attacks with minimum false positives, zero false-negative rate, and low runtime overhead.

Keywords Cross-site scripting (XSS) · Feature injection · JavaScript code injection · Same origin policy (SOP) · Sanitization routine · Script injection vulnerabilities

1 Introduction

In the modern era of World Wide Web (WWW), Web applications are well thought-out to be the front-end that provides access to numerous online services (e-mail, social networking sites, Internet banking, e-commerce front-ends, etc.). In our daily routine life, we interact with several Web applications which are developed using a wide variety of different implementation languages such as JavaScript [1], ASP.net [2]. These Web applications can be accessed through diverse varieties of modern Web browsers and provide various different scripting languages backgrounds. Such Web applications also make it tricky for the developers of the Web application to appropriately protect them and remain up-to-date with the rising vulnerabilities and newly revealed attacks.

Usually, attackers have to discover and make use of weakness in a genuine Web site in order to get the access and insert their poorly written script in the Web applications. In order to compromise the security of a genuine Web site, it may appear to be a difficult task for countless. However, vulnerability assessment of unrestricted Web sites achieved in year 2013 by Symantec's Web site Vulnerability Assessment Services [3] discovered that nearly 77 % of Web applications hold vulnerabilities. From this, 16 % were categorized as

✉ B. B. Gupta
gupta.brij@gmail.com

[1] National Institute of Technology, Kurukshetra, India

security-critical vulnerabilities that could permit the attacker to poison the sensitive data (e.g., cookies) or compromise other critical resources. Moreover, it was highlighted in the 2014 Symantec Internet Security Threat Report [3] that 1 out of 8 Web sites had critical vulnerabilities. Moreover, this report also highlights in year 2014 that cross-site scripting (XSS) vulnerability was discovered in wind turbine control application.

Although XSS was discovered in the year 2000 [4], this category of vulnerability created a severe concern. In 2005 only, almost 167 XSS vulnerabilities had reported to the BugTraq mailing catalog. Later on, XSS proxy tool [5] is proposed for the organized exploitation of XSS vulnerabilities. In addition, in October 2005, the "Samy XSS worm" [6] tainted 100,000 user profiles worldwide. Moreover, XSS attacks have turn out to be a XSS attack vectors covering a wide variety of attacks and can be utilized by the attackers to craft a distributed botnet over the network.

The escalation of JavaScript-based browser-side programming has given growth to numerous security problems interrelated to modern Web applications. According to the survey done by the OWASP [7] in 2013, the most infected problem is Cross-Site Scripting (XSS) attack and it is the third top most weakness among the top ten susceptible threats. Figure 1 shows widespread vulnerabilities discovered in Web applications tested by the IBM Hosted Application Security Management(HASM) service, contrast to the OWASP top 10 vulnerabilities for 2013 [7]. The OWASP is a community to facilitate the organizations for developing and maintaining the modern Web applications that can be trusted. Abstract view of exploitation of XSS vulnerability on Web application is shown in Fig. 2. Figure 3 shows the list of OWASP top 10 vulnerabilities in 2013 found in the modern Web applications with its associated description. Figure 1 clearly shows that almost 17 % of the Web applications are susceptible to XSS attacks. Users across social networking sites like Twitter, Facebook are all infected with the malicious XSS payloads.

The trouble of XSS occurs from malicious JavaScript code that can be inserted into a HTML document through untrusted input [8–10]. In addition to this, the improper sanitization and input validation of the data provided by the user of Web application are the main causes of XSS attack. The word "Cross-Site" refers to the security boundaries that the client-side Web browser places on the sensitive user data (e.g., cookies) linked with the Web application. Therefore, if a malicious inserted script is executed within the environment of trust of the Web application's site, attacker can then circumvent the security boundaries which lead to session hijacking, cookies theft, botnet spreading, etc.

1.1 Contributions

The main contributions of this paper are as follows:

XSS-SAFE We propose a novel server-side solution Cross-Site Scripting Safe Web Application FramEwork (XSS-SAFE) which discovers the XSS vulnerabilities in Web applications by injecting the features of benign JavaScript code and prevents this vulnerability by exploiting the automated placement of sanitizers in the injected JavaScript code.

Feature Injection We injected the feature statements incorporating the features of benign JavaScript code and randomly generated tokens. As soon as the HTTP response Web page is initially generated at the Web server side, JavaScript code with no or incorrect feature statement will be considered as injected or malicious JavaScript code.

Sanitization Routines Placement After the successful detection of injected JavaScript code by introducing the concept of feature injection, we have also incorporated the idea of

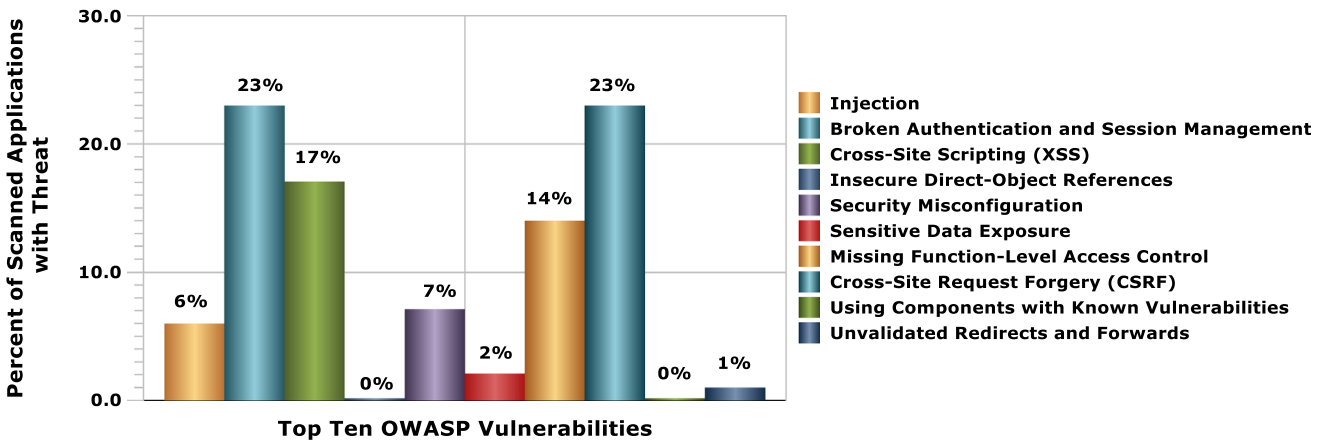

Fig. 1 Common vulnerabilities found occurring in Web applications tested by the IBM Hosted Application Security Management (HASM) service, compared to the OWASP top 10 for 2013

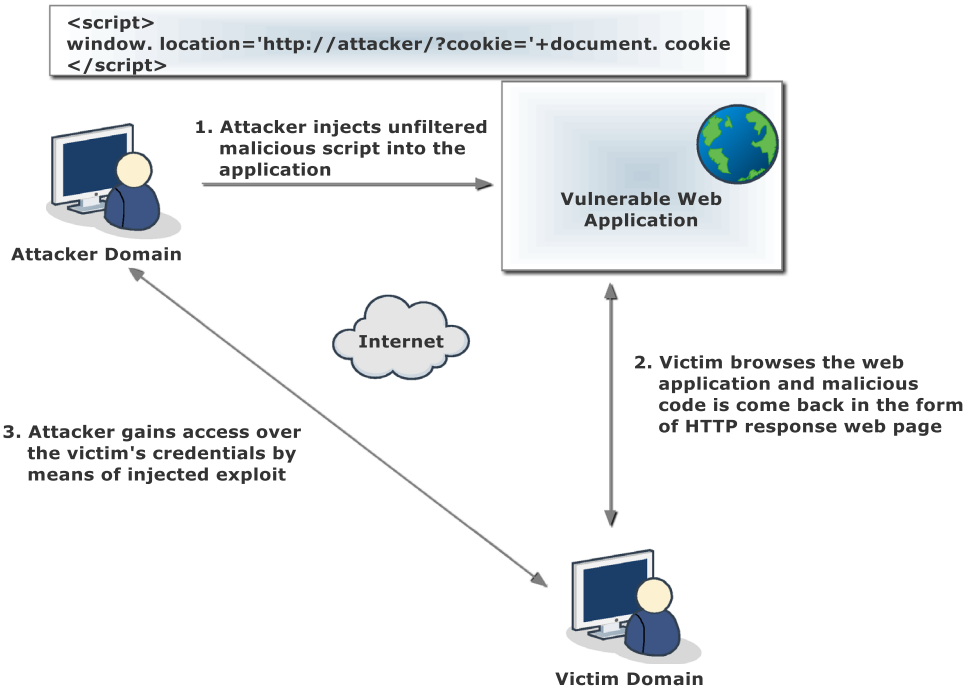

Fig. 2 Abstract view of exploitation of XSS vulnerability on Web application

automated placement of sanitizers in the code of JavaScript. This technique has proved to be helpful in detecting the malicious injected code of JavaScript code before presenting any final HTTP response to the Web browser.

Implementation and Evaluation We evaluate the XSS-SAFE on five server-side programs developed in JSP. We have designed a technique to inject the feature statements of JavaScript and the automated placement of sanitizers in the source code of JavaScript. We have evaluated the XSS-SAFE approach on five real-world JSP programs (JAuction, JVote, JInsure, MeshCMS, and EasyJSP) and found that XSS-SAFE can discover a subset of malicious injected JavaScript code with minimum false positives, zero false-negative rate and observed low runtime overhead.

Rest of the paper is organized as follows. Section 2 contains overview, motivation, and challenges to deal with XSS attacks. Related work is presented in Sect. 3. Proposed solution is discussed in detail in Sect. 4. Implementation and evaluation results are presented in Sect. 5. Finally, Sect. 7 concludes our paper and discusses scope for future research.

2 Overview, Motivation, and Challenges

Cross-Site Scripting (XSS) is known as a leading security problem currently faced by the Web application developers. It is also the most general attack that attackers exploit to replicate the malicious code to victim's Web application [11–14]. Figure 2 illustrates the abstract view of exploitation of XSS vulnerability on the victim's Web application. XSS involves the insertion of a malicious code into a victim's Web application so that in future, when a victim browses the Web application, the malicious script code is executed by the browser of the victim. XSS is a malicious attack vector that is escalating exponentially in prominence. This is so because with the growing nature of the social networking Web sites, more and more, Web sites are permitting the Web users to upload the information to Web applications. There are numerous kinds of XSS repositories (e.g., XSSed.com [15]), which are a proof that Web sites are constantly exposed to XSS attacks. Even though this particular XSS repository is not precious to the research community, this repository can hardly help in solving real XSS problems. Although the main motivation for the XSS vulnerability lies on the Web server side, the factual exploitation of XSS attack is generally seen on the Web browser. As a result, a Web developer of a Web application has only very incomplete proofs of XSS attacks.

It has been generally seen that input validation is the first defensive mechanism deployed on the server side to mitigate the XSS attacks. But, at the same time, this mechanism is considered to be ineffective when the user-injected content contains feature-rich HTML content. Therefore, it is a difficult task for every defensive security method to only filter out the malicious HTML-injected content and preserve legitimate user-generated HTML content with high reliability. We have also presented in this paper numerous current real-world XSS attacks and examined the reasons for the breakdown of input validation methods in shielding these attacks. There are three categories of Cross-Site Scripting (XSS) attacks: Stored, Reflected, and DOM-based XSS attacks.

Fig. 3 List of OWASP 2013 top ten Web application security risks and its description

These categories of attacks vary from each other in the manner they handle to insert the malicious script code into the Web application and in the manner this script code gets executed. Most of the authors do not take into account DOM-based XSS attacks when they specify the diverse XSS attack types. The motivation of that omission is not only the lesser quantity of happening of that particular type of attack but the unlike nature of the attack itself, While reflected and stored XSS attacks occur due to exposed Web applications. DOM-based attacks are provoked by vulnerabilities of the interpreter of the script used by the Web browser. Reflected and stored XSS attacks take advantage of vulnerabilities which are discovered on the Web applications. These attacks inject vulnerable script code through an HTTP request in the form of a parameter or input of a Web form. In reflected XSS attacks, the inserted malicious script is straight away executed in the context of victim's Web browser, since the malicious script is incorporated in the HTTP response of the corresponding HTTP request. On the other hand, stored XSS attack causes the damage in a different way: Their aim is to insert a malicious script permanently on the Web server of the victim's Web application. In this way, an attacker has to make use of vulnerability just one time and the inserted malicious script would execute as many times as the Web page holding the script is visited. Figures 4 and 5 show the pattern scenario of exploitation of stored XSS and reflected XSS attack, respectively.

JavaScript [1] is broadly used scripting language developed by the Netscape [16] to increase the readability and enhancement of client-side display of HTML Web pages. It was afterward standardized by the ECMA Script standard [17]. The ability of JavaScript programs to manage the enhancement and semantic of a Web page is offered by means of global object document which is an indication to the root

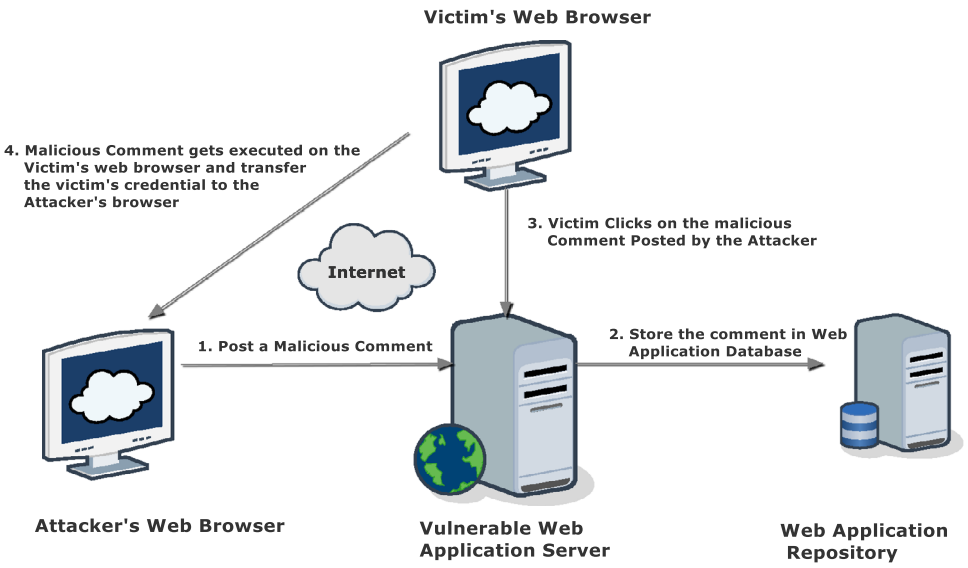

Fig. 4 Exploitation of stored XSS attack

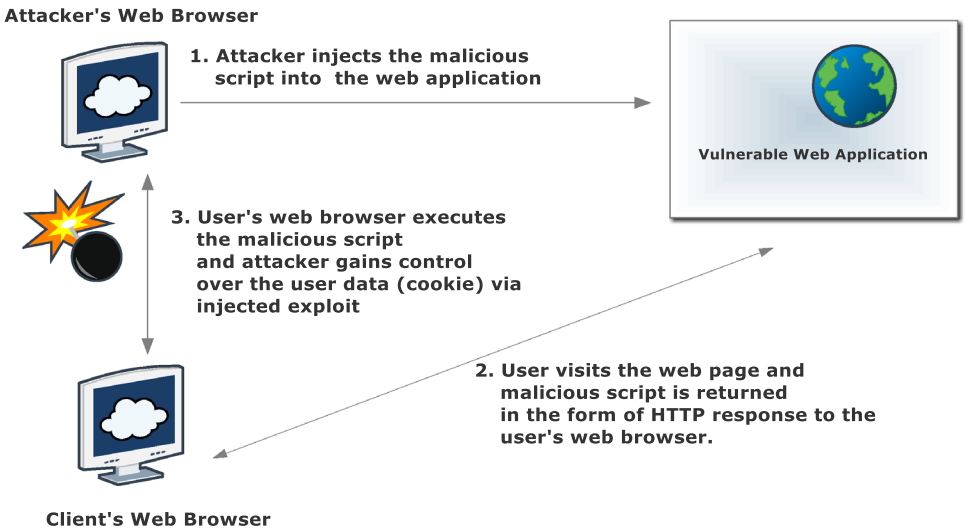

Fig. 5 Reflected XSS attack without violating the same origin policy (SOP)

Fig. 6 Injected script on the vulnerable Web application

```
<a href="http://www.trustedwebsite.domain/
<script>\document.location=http://www.attacker.domain/?grabcookie.php?=+document.cookie;\
</script>'>Click here to know something interesting</a>.
```

component of the Document Object Model (DOM) [18] tree. JavaScript has two major characteristics: sandbox [18] and same origin policy (SOP) [19].

The sandbox thwarts the code of JavaScript from making the changes in the other parts of the system or other Web browser resources. The source code of the Web applications developed in the JavaScript is executed under the protection of a sandbox. It has no right of access to the Web browser's host system and merely restricted access to the properties and functions of Web browser. Therefore, secure execution of source code of JavaScript is "sandboxed," which permits the source code to execute restricted group of operations. On the other hand, SOP is considered to identify that what are the specified operations, which can be altered within this sandbox environment and how this sandboxed source code can communicate with the external world.

Usually, JavaScript programs are considered as unreliable software programs in the modern era of WWW. Such programs downloaded from wide variety of Web sites are protected from each other using a defensive method known as same origin policy (SOP). This policy simply states that what is accessible by the source code of JavaScript. It simply permits a JavaScript program to read or write access to the properties of contents that have the identical origin as the script itself. The origin of the content is identified by the particular contents of the URL address: Host Name, Port Number, and Protocol Version. However, Port Number and Protocol Version are static in nature.

On the other hand, the SOP is also fragile enough to permit partial cross-domain access. The main drawback of this policy is that JavaScript can manipulate the property of host name to circumvent this policy. Corresponding to this policy, there are merely two alternatives: either "no access at all" or "unrestricted access." Moreover, the functions in two different scripts can be invoked on the same Web page even if the two scripts are from dissimilar domains. Although it does not prevent any data from other domains being requested and loaded into the origin Web page, this request for data can be exploited to transfer the information out to any other arbitrary domain for performing any malicious activity like cookie stealing. Such information regarding cookie will be utilized by an attacker to gain the access to the victim's session. Therefore, each particular XSS concern attacks the whole susceptible Web application. Defensive techniques to efficiently restrict the potential of malicious injected scripts are awkward and complex.

Figure 5 shows the scenario of stealing the cookie by reflected XSS attack without violating the same origin policy (SOP). Here, the attacker injects a malicious script into the vulnerable Web application. However, the client's browser starts browsing a vulnerable Web application and could be trapped into clicking on the following malicious link as shown in Fig. 6. When a client-side Web browser clicks on this link, a HTTP request is transferred by the client's Web browser to the Web server of the vulnerable Web application, demanding the vulnerable Web page, i.e., *grabcookie.php*. The Web server of the vulnerable Web application accepts the HTTP request and ensures that if it has the corresponding Web page or not. When the client's Web browser does not locate the requested Web page, then the Web server returns an error message to the client's Web browser, incorporating the name of the malicious script to indicate that which resource was not found. As a result this malicious script gets executed inside the perspective of trust of the origin of site of the Web application. When the malicious script gets executed, then the cookie finally gets transferred to the attacker's Web browser in the form of a parameter. Therefore, the client's Web browser executes the malicious script within the domain of the trust of vulnerable Web application and permits to transfer the cookies to the attacker's Web browser without violating the SOP.

Therefore, this has motivated us to propose a new framework known as XSS-SAFE (Cross-Site Scripting Secure Web Application FramEwork), which is an automated framework for the detection and mitigation of XSS attacks designed on the idea of combining three techniques: Feature Injector, Rule Extractor, and Sanitization Routine Injector. We created a prototype tool to repeatedly inject the feature content and generate rules for the discovery of XSS attacks. We have evaluated this approach on five real-world JSP programs (JAuction [20], Jvote [21], Jinsure, MeshCMS [22], and EasyJSP [23]).

3 Related Work

By analyzing high impact of XSS attacks on the modern Web, a considerable quantity of research work has been done in the past. Shahriar et al. [24] had designed an automated server-side XSS detection approach, $S^2 \times S^2$, which is developed on the concept of "boundary injection" to encapsulate dynamic-generated content and "policy generation" to validate the

data. But this technique consumes more time in policy checks and thus degrades the attack detection capability. On the other hand, Shahriar et al. [25] had proposed a server-side JavaScript code injection technique which relies on the idea of inserting comment statements consisting of random generated tokens and characteristics of benign JavaScript code. But, it has been observed that their technique discovers only a part of code injection attacks. Some of the code injection attacks are still neglected by their approach. Furthermore, because of the drawbacks of JavaScript parser, there is no automated process shown for the removal of various preprocessing techniques of JavaScript such as comment tags and return keyword in event handlers.

In addition to this, it has also been observed in the past by several researchers that the initial line of guard for XSS attacks is input filtering. Path Cutter is an XSS detection tool, which utilizes the dynamic analysis in order to block the dissemination of unsafe JavaScript through XSS vulnerabilities [26]. The major drawback of their work is the rendering latency observed by Path- Cutter at the Web browser. However, the response time of XSS-SAFE fluctuates between 1.25 and 5.75 %, which is bearable. BIXSAN filters out the malicious HTML content and converts the remaining HTML content into Document Object Model (DOM) [27]. However, this technique considers only a small subset of malicious scripts from XSS cheat sheet [28]. In our work, we have extracted all possible encoded and decoded versions from XSS cheat sheet and test these scripts on XSS-SAFE.

Saxena et al. [29] had discovered client-side input validation vulnerabilities on real-world Web applications, which generally arise due to invalidated usage of untrusted data in JavaScript code. Their work does not handle the complexity of sanitization errors. However, XSS-SAFE focuses on the complexity of sanitization errors, which generally reside in the client-side JavaScript code. Content Security Policies (CSP) generally permit the injection of a security policy via HTTP response headers and meta-tags, which is required to be imposed in the Web browser [30]. This policy can limit the location from where a Web site loads its content, therefore mitigating some of the Web-based content injection attacks. Also, CSP merely provides boundaries on a subset of the security aware procedures, namely content injection vulnerabilities like malicious script injection. However, in our method, we have incorporated a method for making a distinction between the malicious injected features and server-trusted content.

SWAP is generally operated, based on the notion of discovering all static script calls in the Web application and encoded them into syntactically invalid identifiers (script IDs) and therefore will not be executed by the JavaScript detection component [31]. Although SWAP is not succeeded in discovering browser-specific XSS. However, XSS-SAFE is successful in detecting such XSS as our technique relies on comparing the stored and extracted features of JavaScript irrespective of different platforms of Web browsers. Galan et al. [32] proposed multi-agent system which operates by discovering the input locations of the Web applications vulnerable to a stored XSS attack and then inject the chosen malicious attack vectors at the formerly detected locations. Gundy et al. [33] proposed Noncespaces which is twofold technique: Firstly, it discovers malicious content so that the Web browser can use a policy to restrict the abilities of malicious content. Secondly, it thwarts the malicious content from altering the DOM tree. However, Noncespaces has not been successful completely for sanitizing the malicious content. They cannot evade all the troubles and obscurity that occurs with sanitization. XSS-SAFE introduced the concept of automated sanitization routine injector, which sanitizes the malicious code effectively. Agten et al. [34] designed a technique, which facilitates the developer of a Web site to safely incorporate third-party scripts, with no requirement of disorderly alterations to both client- and server-side infrastructure. Normally, Web developers install Jsand by incorporating the Jsand JavaScript library in their Web pages.

Table 1 provides a summary of several related state-of-art techniques on the detection and prevention of XSS attacks on Web applications. The first column highlights the titles of the different related techniques. The second column explains their proposed methodology. The third and fourth columns emphasize on their strengths and limitations, respectively, to identify the research gaps. Currently, there exists almost no technique that deals with the insertion of feature statements of JavaScript code and sanitization routines simultaneously in the code blocks of JavaScript statements for the detection and prevention of XSS attacks.

In addition, Table 2 provides a comparison of the state-of-art techniques described in Table 1 based on nine different categories: exploitation location, discovery site, scrutinizing mechanism, persistent attack detection, non-persistent attack detection, DOM-based XSS attack detection, client-side Web browser alterations, server-side modifications, and JavaScript code amendments. The statistics shown in Table 2 clearly highlights that majority of the XSS defensive mechanism does not shield against DOM-based XSS attacks. Moreover, several techniques demand major alterations at the client Web browser and as well as server side. However, several defense mechanisms incorporate the sanitization mechanisms in the code of JavaScript for shielding against the XSS attacks. But again, majority of these techniques suffer from wide variety of false positives and false-negative rates. However, researchers in the past have focused on these two methods separately that too suffers from various false positives and false negatives. Therefore, in this paper, we have proposed XSS-SAFE, which is designed based on the thought of injecting the features of JavaScript and introduced an idea of injecting the sanitization

Table 1 Summary of related detection and prevention techniques of XSS attacks

State-of-art techniques	Methodology	Strength	Flaws
Path Cutter: severing the self-propagation path of XSS JavaScript worms in social Web networks [26]	Path Cutter blocks two major self-propagation paths of an XSS worms, i.e., illicit HTTP Web request to the Web server from the tainted Web page and illegitimate access to the DOM of victim's Web page	Path Cutter generally jams the transmission path of XSS worms by restricting the DOM access to several different views at the Web browser and hampers the illicit HTTP Web requests to the Web server	This tool cannot block the propagation of Drive-by Download worms. Also, Path Cutter tool is failed to jam the exploitation of Phishing and Click-Jacking Attacks
BIXSAN: Browser Independent XSS Sanitizer for prevention of XSS attacks [27]	BIXSAN retains the static tags and filters out the dynamic tags. BIXSAN sanitizes the injected code and transformed this code into Document Object Model (DOM)	BIXSAN consists of an HTML parse tree producer, which is used to diminish the inconsistent performance of Web browser as well as for the recognition of static script tags	BIXSAN was evaluated by referring the parsing quirks in the XSS cheat sheet [28]. But with the advancement in the technology of JavaScript tags, this cheat sheet is dynamically growing day by day and BIXSAN is unable to detect the XSS attacks on them
FLAX: Systematic discovery of client-side validation vulnerabilities in rich Web applications [29]	Flax, a taint enriched black-box fuzzer, utilizes a hybrid and dynamic analysis methodology, which examine the JavaScript-based Web applications for the discovery of input validation vulnerabilities	This tool is lightweight compared to other techniques, no false positives and is scalable sufficient to test it on numerous open-source real-world Web applications	The testing of FLAX has not focused on the complexity of sanitization errors, which still remain in the client-side JavaScript code
Reining in the Web with Content Security Policy [30]	The authors propose the creation of Content Security Policies (CSP), which tries to diminish the Web-based injection attacks by permitting the Web applications to just execute external scripts from white-listed external JavaScript files	In CSP, a Web server formulates its policy and transfers to CSP-compliant Web browsers, which can utilize it to alleviate attacks generated from Web application vulnerabilities like XSS attacks, Data Leak Attacks, etc.	CSP has been failed in those cases where a JavaScript-enabled Web application has been compromised and is providing malicious JavaScript
SWAP: Mitigating XSS attacks using a reverse proxy [31]	SWAP is a server-side solution for the discovery and mitigation of XSS attacks and vulnerabilities with the help of a mechanism of reverse proxy provided with a Web browser	It has a fine capability of detecting the deviation between benign and injected JavaScript code	Many categories of XSS attacks cannot be detected by this technique
A multi-agent scanner to detect stored XSS vulnerabilities [32]	Such multi-agents survey the Web site in order to locate the possible injection points where stored XSS attacks can be exploited	This multi-agent scanner does not require access to the source code of the scanned application. The architecture is extremely flexible and adjustable for the scanning of any Web application	The performance is very low because of the existence of the exploitation of multi-agent system to discover the malicious JavaScript code. Likewise the recognition rate is 39.8 %

Table 1 continued

State-of-art techniques	Methodology	Strength	Flaws
Noncespaces: Using randomization to enforce information flow tracking and thwart cross-site scripting attacks [33]	Noncespaces is an end-to-end mechanism that facilitates Web browsers to differentiate between benign and malicious content to apply the techniques from Instruction Set Randomization (ISR) for thwarting the exploitation of XSS vulnerabilities	Evade all the troubles and obscurity occurs with sanitization	Does not provide any defensive mechanism regarding inserted JavaScript code downloaded from remote Web site
Jsand: Complete client-side sandboxing of third-party JavaScript without browser modifications [34]	Jsand is a server-driven JavaScript-based sandboxing support which implements a server-specific policy on the injected scripts with no requirement of filtering or modification of scripts	The technique facilitates the developer of a Web site to safely incorporate third-party scripts, with no requirement of disorderly alterations to both client- and server-side infrastructure	This technique does not shield against script-less attacks. Also, it relies on the exploitation of Secure EcmaScript [17], which is incompatible for various JavaScript libraries
Injecting comments to detect JavaScript code injection attacks [25]	It is a server-side JavaScript code injection technique which relies on the idea of inserting comment statements consisting of random generated tokens and characteristics of benign JavaScript code	Their strength relies on discovering the XSS attacks by inserting the comment statements consisting of random generated tokens and characteristics of benign JavaScript code	It has been observed that their proposed technique discovers only a part of code injection attacks
$S_2 \times S_2$: A server-side approach to automatically detect XSS attacks [24]	It is an automated server-side XSS detection approach that is developed on the concept of "boundary injection" to encapsulate dynamic-generated content and "policy generation" to validate the data	This idea of boundary injection identifies probable HTML tags, JavaScript content, and other various expected features, which are analyzed throughout the generation of HTTP response Web page to discover the XSS attacks	Their technique consumes more time in policy checks and thus degrades the attack detection capability

Table 2 Summary of related XSS defensive mechanisms based on specified categories

Related work techniques	Exploitation location	Discovery site	Scrutinizing mechanism	Persistent XSS attack detection	Non-persistent XSS attack detection	DOM-based XSS attack detection	Client-side Web browser alterations	Server-side modifications	JavaScript code amendments
Cao et al. [26]	Web Server or Web Proxy	Web Browser	Active	No	Yes	Yes	Yes	Yes	No
Chandra et al. [27]	Web Server	Web Browser	Active	No	Yes	No	No	Yes	No
Saxena et al. [29]	Web Browser	Hybrid	Passive	Yes	No	No	Yes	No	No
Stamm et al. [30]	Web Browser	Web Browser	Active	No	Yes	No	Yes	No	No
Wurzinger et al. [31]	Web Proxy or Server	Web Server	Passive	Yes	Yes	No	Yes	No	Yes
Galan et al. [32]	Web Applications	Web Browser	Active	Yes	No	No	No	No	No
Gundy et al. [33]	Web Server	Hybrid	Active	Yes	Yes	No	No	Yes	No
Agten et al. [34]	Web Server	Web Browser	Active	Yes	No	No	Yes	Yes	No
Shahriar et al. [25]	Web Server	Hybrid	Active	Yes	No	No	No	Yes	Yes
Shahriar et al. [24]	Web Server	Hybrid	Active	Yes	No	No	No	Yes	Yes
XSS-SAFE (Our Work)	Web Server	Hybrid	Active	Yes	Yes	Yes	No	Yes	Yes

routines in the source code of JavaScript to detect and mitigate the XSS vulnerabilities from Web applications. We frequently inject the feature content, generate rules, and add sanitization routines for the discovery of XSS attacks. We have evaluated this approach on five real-world JSP programs. The results indicate that XSS-SAFE detects and mitigate most of the previously known and unknown XSS attacks with minimum false positives, zero false-negative rate, and low runtime overhead.

3.1 Comparison of XSS-SAFE

We have proposed XSS-SAFE motivated by the two recent related techniques [24,25]. Our work is almost similar to their work. Therefore, we have compared the observed results of XSS-SAFE with these two related techniques which are listed in the following Table 3. The outcomes observed in the results of XSS-SAFE indicate the high overall novelty beyond these two prior related techniques.

4 Proposed Solution: XSS-SAFE (Cross-Site Scripting Safe Web Application FramEwork)

In this section, we have discussed our proposed novel solution, i.e., XSS-SAFE, a server-side approach for discovering the XSS attack vectors in the HTTP response messages and tries to mitigate them with minimal false-positive rate. XSS-SAFE operates in two different phases: training and detection phases. The detailed algorithm of these two phases is as shown in Tables 4 and 5.

Figure 7 highlights the abstract view of XSS-SAFE framework. The Web server receives the HTTP request from the Web browser. The Web server is provided with the customized code by inserting the features in the form of comments. As a result, when the HTTP response is generated at the server side, it will be checked by the sanitizer for detecting any variation between the observed and stored features. Any type of variation will be considered as a malicious injection by the attacker, and an alert message will be transferred to the client's Web browser. Otherwise the final HTTP response is transmitted to the Web browser. The detailed design of XSS-SAFE framework is discussed in Fig. 8.

4.1 XSS-SAFE Framework

Description of various modules of XSS-SAFE is mentioned below:

Code Tracer This module is responsible for generating the Control Flow Graph (C_{FG}) from each module of the Web application. It consists of three main subcomponents: lexi-

Table 3 Comparison of XSS-SAFE with the recent related techniques

Factors	Injecting comments to detect JavaScript code injection attacks [24]	$S_2 \times S_2$: a server-side approach to automatically detect XSS attacks [25]	XSS-SAFE (our work)
Inclusion of legitimate inputs	Their work may or may not include the benign HTML input to be a part of user input	This technique permits the legitimate HTML content as a part of user input	XSS-SAFE does not include the legitimate HTML inputs since it can further produce the possibility of including remote JavaScript file in an anchor tag
Detection of malicious JavaScript method calls	This approach can be evaded for certain JavaScript method calls which are almost identical to legitimate calls	$S_2 \times S_2$ could be circumvented for certain specific XSS attacks	XSS-SAFE has the ability to discover the malicious JavaScript method calls which are similar to benign JavaScript code
Inclusion of features of JavaScript code and sanitization routines	The presence of duplicate comments injected by the attacker can be simply circumvented by their technique as they have not employed the sanitization routines in the source code of JavaScript for detecting such duplicate comments	This technique depends on legitimate JavaScript code features and does not apply any sanitization routine in the source code of JavaScript for detecting any malicious code	We repeatedly inject the feature content, generate rules, and insert sanitization routines for the discovery of XSS attacks
Automated preprocessing	This technique requires manual preprocessing of elimination of injected comments from the source code of JavaScript	It also requires manual preprocessing for removal of return keywords in event handlers and injected comments from the source code of JavaScript	We have introduced the automated concept of injection and extraction of features and sanitization routines from the source code of JavaScript
XSS attack discovery proficiency	The approach consumes more time in policy checks and suffers from a large runtime overhead	It also consumes more time in feature comparison and policy checking	Our technique consumes tolerable time as we have introduced an automated process of injection of features and sanitization routines

Table 4 Training phase of XSS-SAFE

Algorithm: training phase	
1.	Generate the C_{FG} from each possible module M_{JSP}
2.	Retrieve the R_A and input the value of R_A to the Web spider component
3.	Crawl the Web pages in such a way that the number of Web pages to be crawled should not exceed the W_{Max}
4.	Extract the contents of JS_C in three possible categories of scripts: automated scripts, event handlers, and URI link scripts
5.	Append the features F_i (for $i = 1, 2, 3, \ldots, n$) in the contents of JS_C with some random value of token R_i (for $i = 1, 2, 3, \ldots, n$)
6.	Store these values of F_i in the attack discovery rules component, which will be further utilized by the HTTP response deviation detector component for discovering the deviation in the content of features
7.	Finally, this instrumented code is stored on the Web server

C_{FG}: control flow graph, M_{JSP}: module of JSP Web application, R_A: root address of C_{FG}, W_{Max}: maximum limit of Web pages, JS_C: JavaScript code, F_i: features of JS_C

Table 5 Detection phase of XSS-SAFE

Algorithm: detection phase	
1.	Extract the initial $HREQ_i$ from the Web browser
2.	Transmit the corresponding $HRES_i$ to the HTTP response deviation detector component for discovering the variation between the BJS_C and IJS_C. Any deviation observed in the extracted set of features ($F_1, F_2, \ldots F_n$) will be considered as an injected JS_C
3.	Remove all the features F_i (for $i = 1, 2, 3, \ldots n$) from the contents of JS_C
4.	Inject the S_R for the BJS_C as well as IJS_C in a static manner by utilizing several encoded patterns of JS_C (as shown in Table 3 and 6)
5.	The JavaScript sensor component will now detect the contents of IJS_C. If IJS_C bypass the automated sanitization routine injector then an alert message will be transmitted to the Web browser
6.	Otherwise, the $SANJS_C$ will be transferred to the sanitization routine extractor, which will remove all the S_R from the BJS_C. The resultant $HRES_i$ is now transferred and interpreted safely by the client-side Web browser

$HREQ_i$: HTTP request, $HRES_i$: HTTP response, BJS_C: benign JavaScript code, IJS_C: injected JavaScript code, S_R: sanitization routines, $SANJS_C$: sanitized JavaScript code

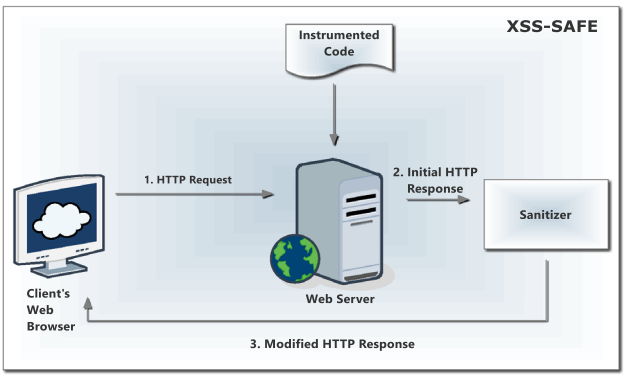

Fig. 7 Abstract view of XSS-SAFE

cal analyzer, parser, and control flow graph generator. The code tracer will utilize the capabilities of lexical analyzer and parser for the generation of C_{FG}. We have also exploited the capabilities of Jflex [35] and Java Parser Cup [36] for the construction of a C_{FG}.

Web Spider A Web spider is a quite simple mechanized program that systematically scrutinizes or "crawls" through the Web pages to generate a directory of the information. To initiate a crawling session, the Web spider component of this framework needs to be input with the origin Web address. By means of this starting Web address, the Web spider moves down the hierarchy of the link tree, gathering all Web pages and incorporated Web forms throughout this procedure. Similar to a typical Web spider, the XSS-SAFE has also configurable alternatives for traversing the utmost link tree depth, upper limit of number of Web pages per domain to crawl, time limit of execution of maximum crawling, and corresponding alternatives for reducing out side Web page links. During this crawling stage, the Web spider component is seeded with the crawling jobs for every Web page that is to be scrutinized for referring target links and possible user-input Web forms.

JavaScript Extractor This component will analyze the source code of Google Chrome Web browser not only for extracting

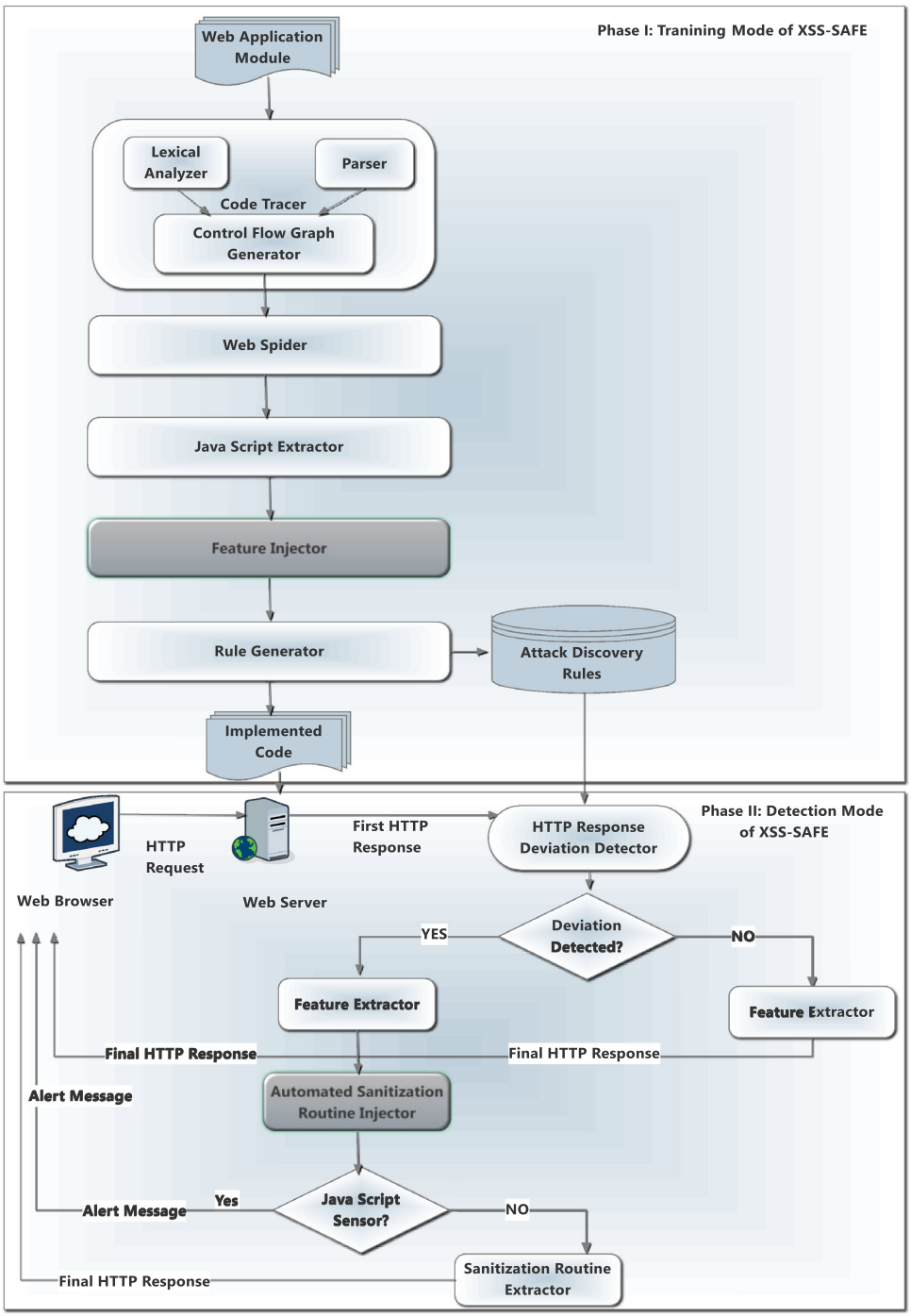

Fig. 8 Design of XSS-SAFE framework

the script content but also for determining all possible numerous ways of embedding JavaScript into an HTML Web page.

- Initially, the JavaScript extractor component will search for those scripts which are executed in an automated manner on loading of the Web page.
- Secondly, this component will discover the event handlers, which will only execute on user interaction.
- Lastly, this component detects the JavaScript URL link scripts, which will again be executed on a user click.

Therefore, these are the three possible ways of discovering ways of injecting JavaScript code into an HTML Web document. The success of this JavaScript Extractor component depends on the degree of coverage of the source code of Web application achieved.

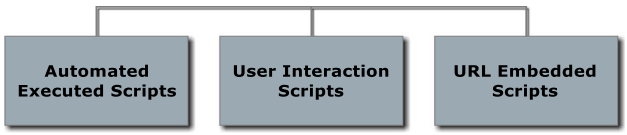

Fig. 9 Three JavaScript types

Feature Injector Feature injection is the process of inserting an HTML or JavaScript comment or feature that does not modify the intended HTTP response or behaviors. The features are evaluated to discover the existence of malicious injected contents. In order to differentiate between the benign and injected JavaScript, XSS-SAFE framework introduces the concept of feature injection which includes random generated tokens and some of the features of benign JavaScript code (e.g., number of HTML/JavaScript tags, attributes). Whenever HTTP response Web page is produced, JavaScript source code with no or false feature is treated as injected or malicious code. Also the legitimate features are verified for duplicity.

The existence of duplicate features or any type of variation observed between the likely code features and practically observed code features implies source code of JavaScript as injected or malicious one. For insertion of features at the margin of each block of legitimate JavaScript code, we have identified the areas for the detection of JavaScript content. Figure 9 shows the three different categories of scripts which need serious attention.

Automated Executed Scripts comes under those categories of JavaScript code which executes automatically on the loading of a Web page. User-Interaction Scripts comes under the category of event handlers, which only executes on user interaction. Lastly, URL Embedded Scripts only executes when clicked upon a JavaScript URL link. We reflect on several general cases where XSS attack vectors might be present:

- Inline Scripts,
- Script Insertion Using Local Source Files,
- Script Insertion Using Remote Source Files,
- Event Handler Code, and
- URL Attribute Values.

Rule Generator This component is responsible for extracting the expected content features and saving these features in the form of some well-defined rules. This module is responsible for generating some well-defined rules based on the injected features and stores these rules in the attack discovery rules module.

Attack Discovery Rules In this phase, we recognize the probable features of the source code of JavaScript, storing these features in attack discovery rules in the form of policies. These policies will be referred by the HTTP response deviation detector component to discover any possible deviation between the expected JavaScript code features and the observed features.

HTTP Response Deviation Detector This component will receive the initial HTTP response Web page and differentiates between the benign and injected/malicious JavaScript code by referring the defined policies in the attack discovery rules. Any deviation observed between the expected and observed code features will be considered as injected or malicious JavaScript code. During this phase, various checks are executed by the deviation detector to discover the injected or malicious JavaScript code. These consist of following:

- JavaScript code without injected feature statement,
- JavaScript code with accurate and redundant feature statement, and
- JavaScript code with non-redundant feature statement.

The HTTP response deviation detector component scans the features of JavaScript produced between the two margins for determining whether the scanned features signify XSS attacks or not. But, to resolve this issue, we have obtained the intended features of JSP expression and relate these JavaScript features via explicit locations of the program with the purpose of discover XSS attacks in the advanced stage. This problem is resolved by examining program source code through an appropriate parser (Jericho [37]) which identifies the tags of HTML as well as produces Document Object Models (DOMs) from the Web pages of JSP.

The expressions of JSP are obtained by creating the values in the perspective of wide variety of HTML outputs like tag text and attribute values. We recognize extracted content as a portion of mistrusted attack input, if this attack input disregards the intended features of JavaScript constructed on explicit output generation perspective. An expression of JSP is positioned between <td>and </td>, and therefore, it must not include an extra tag. If the expression is utilized to adjust a value of attribute, then the created content must not include any fresh attribute. Injection of features all over the expressions of JSP as well as verification of the existence of HTML tags could not recognize such contexts in an exact manner.

We validate this concern with two illustrations as shown in Table 6. The first two columns consists of td and id attribute which is allocated by means of the dynamic contents (<%=name%>). Third column demonstrates insertion of features surrounding the expressions of JSP. We display the legitimate inputs as well as the outcomes produced during the parsing of contents between features in the fourth and fifth columns. The legitimate inputs comprise Bob and id1. The fifth column displays the outcomes in relation to the amount of attributes and tags. We detect legitimate inputs, which are

Table 6 Comment injection around JSP expressions and analysis results

Context type	JSP expression	JSP expression with comments	Legitimate test input	Result (Tag, Att)	Attack input	Result (Tag, Att)
Tag text	<td> <%=name%> </td>	<td> <!--t1--> <%=name%> <!--t1--> </td>	Bob	(0,0)	<script>alert("XSS"); </script>	(1, 0)
Attribute value	<td id = "<%=name%>" </td>	<td id = "<!--t1--> <%=name%> <!--t1-->" > </td>	id1	(0,0)	id1" onfocus = "foo()"	(0,0)

Table 7 Comments around HTML tags and analysis results

Context type	Comment	Expected (Tag, Att)	Attack input	Result (Tag, Att)
Tag text	<!-- t1--> <td> <%=name%> </td> <!--t1-->	(1, 0)	<script>alert("XSS");</scr ipt>	(2, 0)
Attribute value	<!-- t1--> <td id = "<%=name%>" > </td> <!--t1-->	(1,1)	id1" onfocus= "foo()"	(1,2)

effectively parsed as well as produce no tag if we inject the features surrounding the expressions of JSP. The last two columns display vectors of attack input as well as equivalent results, separately. The attack inputs comprise insertion of a script tag and an id1 attribute value with a random function call foo. By examining the parsing outcomes of attack vector inputs, we observe that the initial example produces a tag. As the second attack vector input is not initiated with a tag, the inserted value and attribute pair is not detected via parser. Though, the parser could identify the inserted name value pair, if parsing is executed w.r.t. the nearby predecessor and descendant HTML tags exist in the Document Object Model (DOM) tree (i.e., <td> …</td>).

We represent nearby descendant as well as predecessor HTML parent tags as "instantaneous parent tag." Likewise second example illustrates that calculation of mutual tag and attribute is significant for recognizing indirect input attack vectors. We refer the quantity of tags and attribute totals w.r.t. the instant parent tag of a JSP expression as probable output features that must retain during runtime. Table 7 illustrates that the proposed approach XSS-SAFE can efficiently discover malicious contents. The second column illustrates the features that are injected w.r.t. the instant parent tags of expressions of JSP. The third column displays the intended quantity of tags and attributes w.r.t. instant parent tags recognized via parser.

The fourth and fifth columns display attack vector inputs as well as real features of produced contents. It is perceived that the probable features of output (third column) do not match with the real features of output for contents comprising attack input vectors (last column). Related arguments could be prepared for the tags of JSTL in which instant parent tags of HTML are suitable for recognizing intended features.

Table 8 An example rule and modified comments

Example rule	Boundaries with a rule id
<RuleID>1</RuleID>	<!--t1:1-->
<Type>call</Type>	<td>
<name>alert</name>	<%=name%>
<argcount>1</argcount>	</td>
<value> document.cookie </value>	<!--t1:1-->

We resolve the problem of concerning features of JavaScript with explicit program positions by storing the likely features of dynamic contents as well-defined rules as well as insert it inside the features. An example of well-defined rule related to the tag text example is shown in the XML format in Table 8. It has five fields that comprise RuleID, Type, name, argcount, and value. The *RuleID* is a unique integer value to relate a rule with an output generation position. The Type denotes rule type that can be adjusted to three different values: expr, scriptlet, and JSTL. In this example rule, it is set as an expression of JSP (expr). The name indicates the title of function, argcount (quantity of arguments utilized), and value (values of arguments utilized).

Feature Extractor After executing the module of HTTP response deviation detector, the feature extractor component will remove all the injected features inserted during the phase of feature injection. The main motto of injecting the features in the source code of JavaScript is to discover the existence of malicious injected contents. Now the modified HTTP response Web page is transferred either to the client-

Table 9 List of encoded variants of "<"

Encoding type	Encoded variant of "<"
JavaScript Hexadecimal Escape Sequence	\x3C
JavaScript Single Escape Character	\<
JavaScript Unicode Escape Sequence	\u003c
URL Encoding	%3C
HTML Hexadecimal Character Encoding 1	<
HTML Hexadecimal Character Encoding 2	<
HTML Hexadecimal Character Encoding 3	&#X3c
HTML Character Entity Encoding 1	<
HTML Character Entity Encoding 2	&lt
HTML Character Entity Encoding 3	<
HTML Character Entity Encoding 4	&LT
HTML Decimal Character Encoding 1	<
HTML Decimal Character Encoding 2	<
HTML Decimal Character Encoding 3	<
Universal Transformation Format (UTF-7) Encoding 1	+ADw-
Universal Transformation Format (UTF-7) Encoding 2	+ADx-
Universal Transformation Format (UTF-7) Encoding 3	+ADy-
Universal Transformation Format (UTF-7) Encoding 4	+ADz-

side Web browser for the safe interpretation of Web content or to the automated sanitization routine injector for the automatic placement of sanitizers in the source code.

Automated Sanitization Routine Injector This component performs a completely an automated technique of sanitizer placement by statically analyzing the stream of injected data in the program. However, placement of sanitizer is static and sometimes changes to dynamic wherever required. The main motto of this component is to discover the inputs for sanitization on the program path from the source to the sink. There are many Web development languages which have built-in functions that sanitize the user-supplied input. Like the PHP function "htmlentities" transforms the characters that have particular significance in HTML into their associated HTML entities (e.g., the character "<" can be transformed into "<").

Table 9 shows the list of several encoded variants of "<." In addition to this, Table 10 illustrates some of the patterns of JavaScript code and their encoded variants. After applying this function to a user-input string, the resultant string can be securely transferred to the client-side Web browser because the resultant string can now no longer be inferred by the Web browser. After the transformation of user-input string into some well-defined group of tokens by using "htmlentities" function, the resultant string should meet the following requirements:

- Firstly, the resultant string should not incorporate any valid HTML or JavaScript tags,
- Secondly, it must not include anything that would be understood by the client-side Web browser, and
- Lastly, the process of conversion of transforming the JavaScript tags must be reversible, so that later on the string can rendered by the Web browser.

Feature Injection We have highlighted some of the cases in Fig. 9 where injected JavaScript code may be present. Based on that we have injected the features in the source code of JavaScript, we prefix and suffix the function definitions with the feature statements (/*f1*/). Here, "f1" is sequence of 32 random characters. Table 11 illustrates some of the patterns of injection of features in some patterns of JavaScript code. We consider some of the general cases of JavaScript. Here, we prefix and suffix the definitions of these code with feature statements (/*f1*/). Here, f1 is a sequence of 32 random characters. The third pattern of JavaScript code illustrates an event handler (onload). Here, we prefix and suffix the JavaScript tag with the feature statements (/*f1*/).

JavaScript Sensor This component will detect the injected script code which is not sanitized properly by the automated Sanitization Routine Injector. Therefore in this case, an alert message will be sent to the client-side Web browser. Otherwise, the request will be passed to the next component, i.e., Sanitization Routine Extractor.

Table 10 Encoded patterns of JavaScript code

Pattern of JavaScript code	Encoded JavaScript
 xxs link	<aonmouseover="alert(document.cookie)">xxs link </a&#x3E
<script>alert(document.cookie); </script>	<script>alert(document.cookie); </script&#x3E
<IFRAME SRC="javascript:alert('XSS');"> </IFRAME>	<IFRAME SRC="javascript:alert('XSS');"></IFR& #x41; ME&#x3E

Table 11 Patterns of injecting the features in JavaScript code

Pattern of JavaScript code	JavaScript code with injected feature
XSS link	XSS link
<script>alert(document. cookie); </script>	<script>/*f1*/alert(document.cookie); /*f1*/ </script>
<body ONLOAD=alert(document. cookie)>	<body ONLOAD=>/*f1*/alert(document.cookie) >/*f1*/>
<script>document.location= \"http://w3schools.com?c=\" +(document.cookie);</script>	<script>/*f1*/document.location=\ "http://w3schools.com?c=\"+(document.cookie); /*f1*/</script>
<IMG SRC=javascript:alert("XSS")>	<IMG SRC=javascript: /*f1*/alert("XSS")/*f1*/>

Table 12 Estimated features of function definitions and calls of JavaScript

Type	Pattern of JavaScript code	Estimated feature
Simple method call	example(7,9,6)	[example,3,7,9,6]
User Defined Method (named)	<h1 onclick="changeText(this)'>Clickonthistext!</h1> <script> function changeText(id) {id.innerHTML = "Oops!";} </script>	[changeText,1,id]
User Defined Method (named)	functionmyFunction(p1, p2){ return p1*p2;}	[myfunction,2,p1,p2]
User Defined Method (anonymous)	var x = myFunction(4, 3); function myFunction(a, b){ return a*b;}	[myfunction,2,4,3]
Simple Method Call	var str ="Apple, Banana, Kiwi"; var res = str.slice(7,13)	[slice,2,7,13]
Nested Method Call	document.write(Date())	[document.write,1,Date()]
Simple Method Call	Math.min(0,150,30,20,-8)	[min,5,0,150,30,20,-8]
Simple Method Call	var str ="HELLOWORLD"; str.charAt(0)	[charAt,1,0]
Host Object method	var element = document.getElementById("header"); element.innerHTML = "NewHeader"	[document.getElementById,1,"header"]
Nested Method Call	Math Floor(MathRandom() *11)	[floor,1,Math.random(0*11]

Sanitization Routine Extractor This component will remove all the sanitization routines properly, i.e., decode the encoded JavaScript string into its original form which can be interpreted safely by the client-side Web browser.

4.2 Generation of Rules Based on the Injected Features of JavaScript

Recognition of Features Injected JavaScript may perhaps not be significant if function calls are not there. The same outcome can be drawn for injected function description that may dominate the actions or program developer implemented function. Therefore, we deduce an interesting feature of function definitions and call from benign source code of JavaScript. We utilize the capabilities of Rhino [39,40], a JavaScript parser for recognizing the function names, number, and names of arguments passed. Table 12 illustrates some of the patterns of benign function definitions.

The first column specifies the type of method utilized for the corresponding code patterns. The second column illustrates the code patterns for these method types. In the first pattern of the code, we have utilized the simple method call named "example" and has three arguments [example (7, 9, 6)]. Now at this point, we represent the predictable feature as [example, 3, 7, 9, 6]. Here, the first two objects represent the name of the method and number of arguments utilized, respectively. The left over objects are the titles of arguments.

The next following row illustrates a user defined method (named) as "changetext," one argument "id." The method definition is inside the script. So here we represent the predictable feature as [changetext, 1, id]. The fourth row shows an anonymous function with no title of the function. In this scenario, the definition of the function is saved in a variable "x." Therefore, we recognize the name of the immediate variable where the function definition is allocated followed by the quantity of arguments and their names, i.e., [myFunction, 2, 4, 3]. The final column highlights a nested call method Math.floor(Math.random() * 11); the related features are [floor, 1, Math.random()*11].

The first pattern demonstrates an event handler, i.e., on MouseOver (in which cursor shifts over an object or area) with one parameter named document.cookie. We utilize the

expected feature content with five fixed fields: rule id (a unique rule id), feature type (), name (the title of the function), argcount (quantity of arguments utilized), and value (values of arguments utilized). The third column illustrates the altered/modified feature that includes mutually random generated feature-based token and the rule id (i.e., /*f1:1*/). Likewise an example policy is illustrated for the second pattern. Here, the rule id is 2, name of defined method is alert, field type is call, argcount is 1, and the value of the passed argument is document.cookie.

Rule Generation The extorted features of JavaScript are accumulated in rule generator files so that source code of JavaScript present in HTTP response Web pages can be matched next to these observed features to discover inserted XSS attack vectors in the form of injected JavaScript code. We have accumulated the feature content in the shape of XML files. Additionally, features also incorporate the rule information. Table 13 illustrates the pattern code, rule information, and customized features.

5 Implementation and Evaluation Results

We conducted a sequence of experimental test with our prototype implementation to express its capability to discover and thwart the XSS vulnerabilities. To this end, our XSS-SAFE framework was tested on five open-source JSP program Web application modules. We implement a prototype tool in Java. The jsoup: Java HTML parser [38] has been utilized to parse the source files and retrieve the Document Object Models (DOMs) of several parsed Web pages. Jsoup offers APIs to access and alter the DOM. The comment injection in JavaScript code blocks involves the requirement of parsing for the source code of HTML. In order to inject the feature statements, various proper position of the program has been altered like event handler function call inserted with comments. We inject the features in various nodes of the DOM and save the altered DOMs (modified programs) back to the source files. Also, the Rhino parser [39,40] is utilized to interpret the JavaScript code, discover, and pull out the features from the AST (Abstract Sytax Tree). We stored all these attack discovery policies in attack discovery rules component, which will be referred by the HTTP deviation detector component for discovering any injected script. The HTTP deviation detector component will be implemented as a server-side sanitizer.

We find five broadly utilized JSP programs from gotocode.com and sourceforge.net to evaluate our XSS-SAFE technique. Figure 10 illustrates the quantity of various categories of scripts (Local Script Insertion, Event Handler, URL Attribute, and Inline Scripts) utilized in the following JSP programs: Jauction, Jvote, Jinsure, MeshCMS, and Easy JSP. We have investigated that most of the JavaScript code in these JSP programs is found as an event handler attribute value or inline code. In addition to this, these JSP programs utilize a small quantity of local files of code of JavaScript. However, none of these employed JSP programs incorporate remote JavaScript file source.

The number of rules created for these utilized JSP programs differs generally because of the technique of presenting the function call and definitions in these programs. We have selected some of the following feature of JSP programs: Argument Count, Function Call, Function Definition, User Function Call, and Host Function Call. Also, we have recognized the quantity of these features in the employed JSP programs. Figure 11 shows the details of quantity of these features in the utilized JSP programs. We have chosen several rules generated from the JSP programs. But the number of rules may vary depending on the technique of presenting the function call and definitions in these programs.

5.1 Evaluation Results

We install the customized server-side programs with injected features in an Apache Tomcat server configured on Operating system Windows 7. All these customized programs saved the data in the MySQL database. The process on which XSS-SAFE is evaluated is a two-stage process. In the first stage, customized programs (programs with injected features) are executed to insert the code of JavaScript though executing the services that receive the inputs and save the inserted code of JavaScript. The second stage runs the programs to execute the services so that the inserted inputs are accessed from repositories and come into view in the HTTP response Web pages. The following Table 14 shows several identified fields in the utilized JSP programs where we have injected the script content along with its associated features and its corresponding sanitization routines for the detection of XSS attacks.

In order to assess the capability of XSS-SAFE, we injected several XSS attack input vectors identified from the XSS cheat sheet [28], which is an extensively utilized source to test XSS attack finding efficiency. The selected XSS attack inputs differ broadly like insertion of script tags, attributes, inline scripts, and third-party URLs. For every injected test case, we install the customized programs with injected features, enable the server-side HTTP response deviation detector, and check HTTP response Web pages for any injected script. We have made use of Google Chrome, Internet Explorer 8 ,and Firefox 3.6 for evaluating the XSS-SAFE framework. The end results are depicted in Fig. 12. The result shows the number of attacks applied, number of attacks detected, number of rules checked, number of warnings encountered, false-negative and false-positive rate (in %age) on the employed JSP programs.

False-Negative Rate Assessment The results in Fig. 12 clearly signify that the proposed technique XSS-SAFE detects all

Table 13 Patterns of rule generation

Patterns of JavaScript code	Rules generated	Modified feature
XSS link	<RuleID>1</RuleID> <type>call</type> <name>alert</name> <argCount>1</argCount> <value>document.cookie</value>	XSS link
<script>alert(document.cookie); </script>	<RuleID>2</RuleID> <type>call</type> <name>alert</name> <argCount>1</argCount> <value>document.cookie</value>	<script>/*f1:2*/alert(document.cookie); /*f1:2*/</script>
<body ONLOAD=alert(document.cookie)	<RuleID>3</RuleID> <type>call</type> <name>alert</name> <argCount>1</argCount> <value>document.cookie</value>	<body ONLOAD=/*f1:3*/alert (document.cookie) /*f1:3*/>
<script>document.location= \ "http://w3schools.com?c = \"+(document.cookie);</script>	<RuleID>4</RuleID> <type>call</type> <name>document.location</name> <link>http://w3schools.com</link> <argCount>1</argCount> <arg>c</arg> <value>document.cookie</value>	<script>/*f1:4*/document.location=\ "http://w3schools.com?c= \"+(document.cookie);/*f1:4*/</script>
<IMG SRC=javascript:alert("XSS")>	<RuleID>5</policyID> <type>call</type> <name>alert</name> <argCount>1</argCount> <value>"XSS"</value>	<IMG SRC=javascript:/*f1:5*/ alert("XSS")/*f1:5*/>
<INPUT TYPE="IMAGE" SRC="javascript:alert('XSS');">	<RuleID>6</policyID> <type>call</type> <name>alert</name> <argCount>1</argCount> <value>"XSS"</value>	<INPUT TYPE="IMAGE" SRC="javascript:/*f1:6*/alert('XSS')/*f1:6*/;">
<BODY BACKGROUND = "javascript:alert('XSS')">	<RuleID>7</policyID> <type>call</type> <name>alert</name> <argCount>1</argCount> <value>"XSS"</value>	<BODY BACKGROUND="javascript:/*f1:7*/alert('XSS')/*f1:7*/">
<BR SIZE="&{alert('XSS')}">	<RuleID>8</policyID> <type>call</type> <name>alert</name> <argCount>1</argCount> <value>"XSS"</value>	<BR SIZE="&{/*f1:8*/alert('XSS')/*f1:8*/}">
<LINK REL = "stylesheet" HREF = "javascript:alert('XSS');">	<RuleID>9</policyID> <type>call</type> <name>alert</name> <argCount>1</argCount> <value>"XSS"</value>	<LINK REL="stylesheet" HREF="javascript:/*f1:9*/alert('XSS')/*f1:9*/;">

Table 13 continued

Patterns of JavaScript code	Rules generated	Modified feature
<IFRAME SRC = “javascript:alert(‘XSS’);”> </IFRAME>	<RuleID>10</policyID> <type>call</type> <name>alert</name> <argCount>1</argCount> <value>“XSS”</value>	<IFRAME SRC=“javascript:/*f1:10*/ alert(‘XSS’)/*f1:10*/;”></IFRAME>
<TABLE BACKGROUND=“ javascript:alert(‘XSS’)”>	<RuleID>11</policyID> <type>call</type> <name>alert</name> <argCount>1</argCount> <value>‘XSS’</value>	<TABLE BACKGROUND=“javascript:/ *f1:11*/alert(‘XSS’)/*f1:11*/”>
<DIV STYLE=“width: expression(alert(‘XSS’));”>	<RuleID>12</policyID> <type>call</type> <name>expression</name> <argCount>1</argCount> <arg>alert(‘XSS’)</arg>	<DIV STYLE=“width: /*f1:12*/ expression(alert(‘XSS’)/*f1:12*/);”>
<META HTTP-EQUIV=“Set-Cookie” Content =“USERID = <SCRIPT>alert(‘XSS’) </SCRIPT>”>	<RuleID>13</policyID> <type>call</type> <name>alert</name> <argCount>1</argCount> <value>‘XSS’<value>	<META HTTP-EQUIV=“Set-Cookie” Content=“USERID=<SCRIPT>/*f1:13*/ alert(‘XSS’)/*f1:13*/</SCRIPT>”>

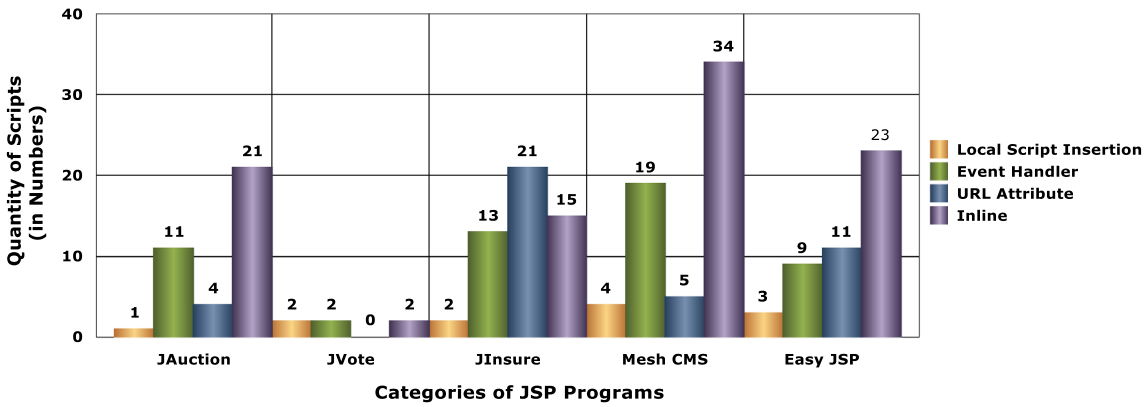

Fig. 10 Quantity of types of JavaScript code in the JSP programs

the injected XSS attack input vectors for every deployed JSP programs. Therefore, the rate of false negative will be 0 % as there is no undetected attack encountered in the result. Since the false negative will be calculated by the ratio between the number of undiscovered attacks to the overall injected attack. *False-Positive Rate Assessment* For computing the false-positive rate, we test the HTTP response Web pages by injecting the legitimate input vectors and search for these legitimate inputs in the generated Web pages. If any of these legitimate inputs are missing in the HTTP response Web page, then it will be considered as false-positive warning. Generally, false-positive rate is calculated by computing the ratio between the total warnings encountered to the total number of rules checked. Figure 12 shows the total number of false-positive warnings, number of rules checked, and false-positive rate for each of these JSP programs. While evaluating the detecting capabilities of XSS-SAFE framework, we encountered some of the false-positive warnings in each of these JSP programs. Also, Fig. 12 clearly indicates that the false-positive rate varies only between 10 and 15 % in all different JSP programs depending on the number of different rules checked.

5.2 Statistical Analysis

In order to prove that the number of XSS attack vectors applied is equal to the number of XSS attacks detected, we use the F hypothesis test, which is defined as:

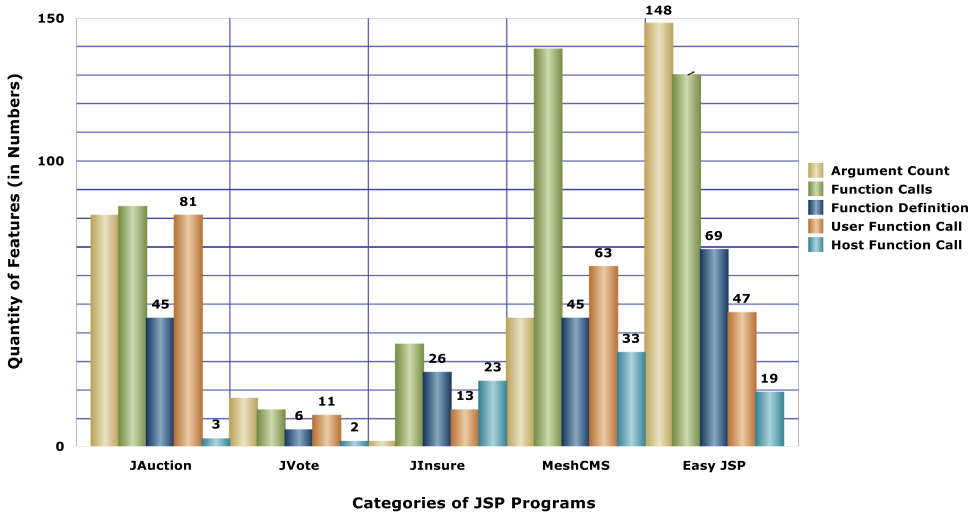

Fig. 11 Quantity of features generated for the JSP programs

Table 14 Summary of several fields of JSP programs

JSP program	Fields
MeshCMS	Create Page
	Edit Configuration
	Edit Map
	Edit Sites
	Edit Source
	Edit User
	Index
	Login
	Main
EasyJSP	Edit Forum
	Edit Member
	Edit Profile
	Login
	Register
JVote	Create Poll
JAuction	Add User
	Add Top Level Category
	Add First Level Category
	Add Auction
JInsure	User Registration
	Agent Registration
	Add Policy

Null Hypothesis H_0 = Number of XSS attack vectors applied is equal to the number of XSS attacks detected ($S_1^2 = S_2^2$).
Alternate Hypothesis H_1 = Number of XSS attack vectors applied is greater than number of XSS attacks detected ($S_1^2 > S_2^2$).

The level of significance ($\alpha = 0.05$). The detailed analysis of statistics of XSS attack vectors applied and detected are illustrated in Tables 15 and 16.

5.3 # of XSS Attack Vectors Applied

of observation (N_1) = 5
Degree of freedom dof (df_1) = $N_1 - 1 = 4$.

6 # of XSS Attacks Detected

of observation (N_2) = 5
Degree of freedom dof (df_2) = $N_2 - 1 = 4$.

$$F_{\text{CALC}} = S_1^2/S_2^2 = 140.6058/140.6058 = 1$$

The tabulated value of F test at $df_1 = 4$, $df_2 = 4$ and $\alpha = 0.05$ is

$$F_{(df1,df2)} = F_{(4,4)} = 6.3882$$

Since $F_{\text{CALC}} < F(4, 4)$, therefore we accept the null hypothesis that the two standard deviations or variation is equal. Hence, it is clear that the number of XSS attack vectors applied is equal to the number of XSS attacks detected, and we are 95 % confident that any difference in the sample standard deviation is due to random error.

Performance Overhead We have calculated some of the HTTP response delays at the HTTP response deviation detector component before sending the actual HTTP response Web page to the Web browser. In order to assess the runtime overhead, we observe the HTTP response Web pages by injecting the input vectors in the existence and nonexistence of HTTP response deviation detector. Table 17 illustrates the review of our findings that incorporates the number of rules checked, response delays (in millisecond) with and without exploiting the capabilities of HTTP response deviation detector and percentage increase in response time. The results showed that the increase in response time fluctuates between 1.25

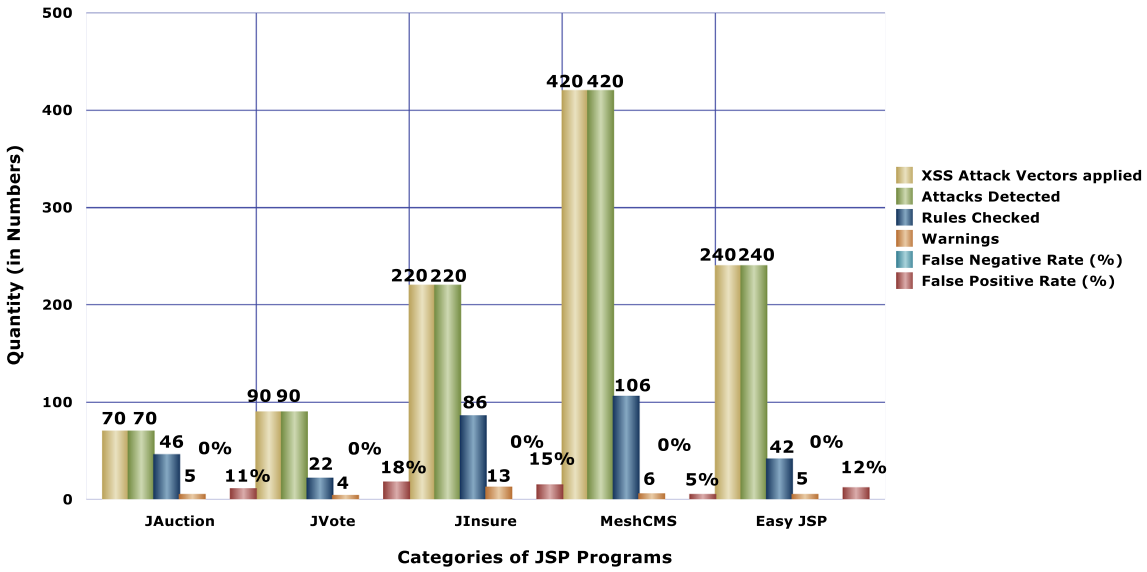

Fig. 12 Overview of observed results

Table 15 Statistics of XSS attack vectors applied

# of XSS attack vectors applied (X_i)	$(X_i - \mu)$	$(X_i - \mu)^2$	Standard deviation S_1 $= \sqrt{\sum_{i=1}^{N_1} (X_i - \mu)^2/(N_1 - 1)}$
70	−138	19,044	140.6058
90	−118	13,924	
220	12	144	
420	212	44,944	
240	32	1024	
Mean (μ) $= \sum X_i/N_1 = 208$		$\sum_{i=1}^{N_1} (X_i - \mu)^2 = 79{,}080$	

Table 16 Statistics of XSS attack vectors detected

# of XSS attack vectors detected (X_j)	$(X_i - \mu)$	$(X_i - \mu)^2$	Standard Deviation S_2 $= \sqrt{\sum_{i=1}^{N_2} (X_i - \mu)^2/(N_2 - 1)}$
70	−138	19,044	140.6058
90	−118	13,924	
220	12	144	
420	212	44,944	
240	32	1024	
Mean (μ) $= \sum X_i/N_2 = 208$		$\sum_{i=1}^{N_1} (X_i - \mu)^2 = 79{,}080$	

Table 17 Results of runtime overhead introduced due to HTTP response delays

JSP programs	No. of rules checked	Response delay without HTTP response Deviation detector (ms)	Response delay in HTTP response deviation detector (ms)	Percentage increase (%)
JAuction	6	2190	2244	2.40
JVote	10	2605	2764	5.75
JInsure	9	2987	3025	1.25
MeshCMS	27	2365	2409	1.82
EasyJSP	17	2987	3098	3.58

and 5.75 %. However, we have recognized that this response delay increases due to the utilization of several numbers of rules checks.

Limitations There is one drawback of XSS-SAFE we would like to point out. XSS-SAFE only detects the relationship between the stored features and injected features in the source code of JavaScript. Therefore, XSS-SAFE cannot mitigate the attacks which bypass the states that exist in the tables of database. If XSS-SAFE is modified to acquire and examine the SQL queries/responses from a database, then it has the potential to handle this category of workflow violation attack. Therefore, this work will be treated as our future work. Our work tolerates the similar drawbacks as other dynamic examination methods. The totality and accuracy of extracted features of JavaScript cannot be assured. We can introduce some manual interference which is desirable to assure adequate training and overwhelm false-positive rate. We would also prefer to examine the methods for automated verification of extracted features.

7 Conclusion and Future Work

XSS vulnerabilities may perhaps be influenced to execute malicious actions by inserting function calls and function definition intervening of JavaScript. Also by analyzing the performance of several related state-of-art techniques, we come to know these server-side code injection vulnerabilities can be evaded by inserting benign code of JavaScript or changing such code vectors downloaded from remote location of Web sites. Moreover, the present server-side XSS vulnerability discovery techniques demand alterations at both client- and server-side locations and transferring of susceptible content from Web server to client-side Web browser.

This paper deal with these issues by introducing a server-side XSS attack detection and prevention solution known as XSS-SAFE based on automated feature injection statements and placement of sanitizers in the injected code of JavaScript. We apply and evaluate the results of proposed approach XSS-SAFE on five real-time JSP programs. The observed result signifies that XSS-SAFE discovers a section of JavaScript code injection vulnerabilities and experienced zero false-negative rate. However, due to the variation in the number of rules utilized for scripts, false-positive rate fluctuates between 10 and 15 %. Moreover, XSS-SAFE facilitates with supplementary advantage of discovering XSS attacks without any modification to client- and server-side commodities.

Future work incorporates discovering the techniques of dropping the HTTP response delay and other rule checks of XSS-SAFE without disturbing its efficiency of XSS attack recognition. Also we will work on decreasing the false-positive rate of XSS-SAFE. We also plan to apply the concept of injection of feature injection statements and sanitization routines in the current HTML5-based Web applications.

References

1. Flanagan, D.: JavaScript: The Definitive Guide, 4th edn. O'Reilly, Newton (2001)
2. MacDonald, M.; Szpuszta, M.: Pro ASP.NET 2.0 in C#," 2005, 1st edn. Apress. ISBN 1-59059-496-7
3. Symantec Corporation. Symantec Global Internet Security Threat Report, vol. 19 (2014)
4. CERT/CC. Cert R _ advisory ca-2000-02 malicious html tags embedded in client web requests. [online]. http://www.cert.org/advisories/CA-2000-02.html (01/30/06), Feb 2000
5. Rager, A.: Xss-proxy. http://xss-proxy.sourceforge.net
6. Samy.: Technical explanation of the myspace worm. http://namb.la/popular/tech.html
7. Open Web Application Security Project (OWASP), http://www.owasp.org
8. Gupta, S.; Gupta, B.B.: PHP-sensor: a prototype method to discover workflow violation and XSS vulnerabilities in PHP web applications. In: Proceedings of the 12th ACM International Conference on Computing Frontiers (CF'15), Ischia, Italy (2015)
9. Gupta, B.B.; Gupta, S.; Gangwar, S.; Kumar, M.; Meena, P.K.: Cross-site scripting (XSS) abuse and defense: exploitation on several testing bed environments and its defense. J. Inf. Priv. Sec. **11**(2), 118–136 (2015)
10. Gupta, S.; Gupta, B.B.: Cross-site scripting (XSS) attacks and defense mechanisms: classification and state-of-the-art. Int. J. Syst. Assur. Eng. Manag. (2015). doi:10.1007/s13198-015-0376-0
11. Klein, A.: Cross site scripting explained. White Paper, Sanctum Security Group, http://crypto.stanford.edu/cs155/CSS.pdf, June 2002
12. Gupta, S.; Sharma, L., et al.: Prevention of cross-site scripting vulnerabilities using dynamic hash generation technique on the server side. Int. J. Adv. Comput. Res. **3**, 49–54 (2012)
13. Gupta, S.; Sharma, L.: Exploitation of cross-site scripting (XSS) vulnerability on real world web applications and its defense. Int. J. Comput. Appl. **14**, 28–33 (2012)
14. Gupta, S.; Gupta, B.B.: BDS: Browser Dependent XSS Sanitizer. Book on Cloud-Based Databases with Biometric Applications, IGI-Global's Advances in Information Security, Privacy, and Ethics (AISPE) series, USA (2014)
15. Cross-Site Scripting (XSS) Attacks Information and Archive. http://xssed.com/
16. Netscape, Accessed from: http://isp.netscape.com/
17. ECMA Script, Accessed from http://www.ecmascript.org/docs.php
18. Sandboxing: https://developer.apple.com/app-sandboxing/
19. Mozilla Corporation. Same origin policy for JavaScript, https://developer.mozilla.org/En/Same origin policy for JavaScript
20. JAuction-0.3, http://sourceforge.net/projects/jauction/
21. Jvote, Accessed from http://sourceforge.net/projects/jspvote/
22. MeshCMS: http://cromoteca.com/en/meshcms/
23. Easy JSP Forum, http://sourceforge.net/projects/easyjspforum
24. Shaihriar, H.; Zulkernine, M.: S2XS2: a server side approach to automatically detect XSS attacks. In: Ninth International Conference on Dependable, Automatic Secure Computing, IEEE, 2011 pp. 7–17
25. Shahriar, H.; Zulkernine, M.: Injecting comments to detect javascript code injection attacks. In: Proceedings of the 6th IEEE Workshop on Security, Trust, and Privacy for Software Applications, Munich, Germany, July 2011, pp. 104–109

26. Cao, Y.; Yegneswaran, V.; Possas, P.; Chen.: Pathcutter: severing the self-propagation path of xssjavascript worms in social web networks. In: Proceedings of the 19th Network and Distributed System Security Symposium (NDSS), San Diego, CA, USA (2012)
27. Chandra, V.S.; Selvakumar, S.: Bixsan: browser independent XSS sanitizer for prevention of XSS attacks. ACM SIGSOFT Softw. Eng. Notes **36**(5), 1 (2011)
28. Rsnake. XSS Cheat Sheet. http://ha.ckers.org/xss.html (2008)
29. Saxena, P.; Hanna, S.; Poosankam, P.; Song, D.: FLAX: systematic discovery of client-side validation vulnerabilities in rich web applications. In: NDSS (2010)
30. Stamm, S.; Sterne, B.; Markham, G.: Reining in the web with content security policy. In: ACM Proceedings of the 19th International Conference on World Wide Web, WWW '2010, New York, NY, USA, pp. 921–930
31. Wurzinger, P.; Platzer, C.; Ludl, C.; Kirda, E.; Kruegel, C.: SWAP: Mitigating XSS attacks using a reverse proxy. In: ICSE Workshop on Software Engineering for Secure Systems. IEEE Computer Society (2009)
32. Galan, E.; Alcaide, A.; Orfila, A.; Blasco, J.: A Multi-agent Scanner to Detect Stored—XSS Vulnerabilities? In: IEEE International Conference on Internet Technology and Secure Transactions (ICITST), June 2010, pp. 332–337
33. Gundy, M.V.; Chen, H.: Noncespaces: using randomization to enforce information flow tracking and thwart cross-site scripting attacks. In: Proceedings of the 16th Annual Network and Distributed System Security Symposium (NDSS), San Diego, CA, Feb 8–11 (2009)
34. Agten, P.; Acker, S.V.; Brondsema, Y.; Phung, P.H.; Desmet, L.; Piessens, F.: Jsand: complete client-side sandboxing of third-party JavaScript without browser modifications. In: Zakon, R.H. (ed.) ACSAC. ACM, pp. 1–10 (2012)
35. Jflex—The Fast Scanner Generator for Java: http://www.jflex.de/
36. Java Parser Cup: http://www.cs.princeton.edu/~appel/modern/java/CUP/
37. Jericho HTML parser, Accessed from http://jericho.htmlparser.net/
38. Jsoup: Java HTML Parser: http://jsoup.org/
39. Rhino Parser, Accessed from http://www.mozilla.org/rhino
40. Rhino, Accessed from http://www.mozilla.org/rhino