Multimed Tools Appl
DOI 10.1007/s11042-016-3735-1

XSS-secure as a service for the platforms of online social network-based multimedia web applications in cloud

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

Received: 23 April 2016 / Revised: 30 May 2016 / Accepted: 27 June 2016

Abstract This article presents a novel framework XSS-Secure, which detects and alleviates the propagation of Cross-Site Scripting (XSS) worms from the Online Social Network (OSN)-based multimedia web applications on the cloud environment. It operates in two modes: training and detection mode. The former mode sanitizes the extracted untrusted variables of JavaScript code in a context-aware manner. This mode stores such sanitized code in sanitizer snapshot repository and OSN web server for further instrumentation in the detection mode. The detection mode compares the sanitized HTTP response (HRES) generated at the OSN web server with the sanitized response stored at the sanitizer snapshot repository. Any variation observed in this HRES message will indicate the injection of XSS worms from the remote OSN servers. XSS-Secure determines the context of such worms, perform the context-aware sanitization on them and finally sanitized HRES is transmitted to the OSN user. The prototype of our framework was developed in Java and integrated its components on the virtual machines of cloud environment. The detection and alleviation capability of our cloud-based framework was tested on the platforms of real world multimedia-based web applications including the OSN-based Web applications. Experimental outcomes reveal that our framework is capable enough to mitigate the dissemination of XSS worm from the platforms of non-OSN Web applications as well as OSN web sites with acceptable false negative and false positive rate.

Keywords Cloud security · Cross-site scripting (XSS) worms · Online social networking (OSN) security · Web security · JavaScript code injection attacks · Sanitization routines

1 Introduction

In the contemporary era of cloud computing, cloud security has turned out to be a serious issue, as numerous on-demand resources are being offered by utilizing the virtualization

✉ B. B. Gupta
gupta.brij@gmail.com

[1] National Institute of Technology Kurukshetra, Kurukshetra, India

Published online: 06 July 2016

technologies of cloud services [1]. Instead of referring the outdated Internet settings for constructing an expensive setup, numerous commercial IT organizations are accessing the services of Online Social Networking (OSN) sites (such as Twitter, Facebook, Linkedin, etc.) on the cloud platforms. In the modern era of Web 2.0 technologies and HTML5-based multimedia web applications, OSN is considered to be the most popular method for information sharing has drawn most of public attention. However, it is clearly known that the cloud settings are installed on the backbone of Internet. Therefore, numerous web application vulnerabilities in the conventional Internet infrastructures also exist in the backgrounds of cloud-based environments.

1.1 Background on XSS attack

The most prominent attack found on OSN-based multimedia sites is the Cross Site Scripting (XSS) attack [11, 13]. Such worms steal the sensitive credentials of the active users by injecting the malicious JavaScript code in the form of some posts on such web applications [10]. Worms that exploit XSS vulnerability are present in almost 80 % of the cloud-based OSN Web applications. Sharing of personal as well as professional information on OSN platform attracts the attackers to launch the attacks like XSS. Table 1 highlights the incidents XSS attack on numerous platforms of OSN. XSS attack is an assault against the Web application wherein an attacker introduces the malicious JavaScript code into the platform of the cloud-based OSN Web applications [12, 14]. This is done in an attempt to theft the user's login credentials and any other sensitive information like session tokens or financial account information. This attack may lead to severe consequences like cookie stealing, account hijacking, misinformation, Denial-of-Service (DOS) attack and many more. Figure 1 highlights the injection of XSS worm on the OSN web server deployed in the virtual machines of cloud platforms.

Here, the attacker injects the untrusted JavaScript on the cloud-based OSN web server. Now, the client's web browser makes a HTTP request (HREQ) for the malicious JavaScript code on the OSN server deployed in the virtual machine of cloud platform. The OSN server responds to the client's web browser with this malicious code in the form of a HTTP response (HRES). The malicious JavaScript code will get executed on the client's domain. Finally, the XSS attack gets exploited and the credentials of the client (such as cookie, credit card number, etc.) will get transferred to the attacker's domain.

Table 1 Incidents of XSS attacks on OSN

Sr. no.	Victim	Year	Category of XSS worm
1.	UK Parliament Web Site	2014	Persistent XSS
2.	Video Sharing Website	2014	Non-Persistent XSS
3.	Yahoo! Mail	2013	Persistent XSS
4.	Paypal Website	2012	Persistent XSS
5.	Facebook	2011	Non-Persistent XSS
6.	Hotmail Website	2011	Persistent XSS
7.	eBay Website	2011	Persistent XSS
8.	Twitter Website	2010	Mikeyy [21]
9.	Orkut Website	2009	XSS bug [21]
10.	MySpace	2006	Samy [20]

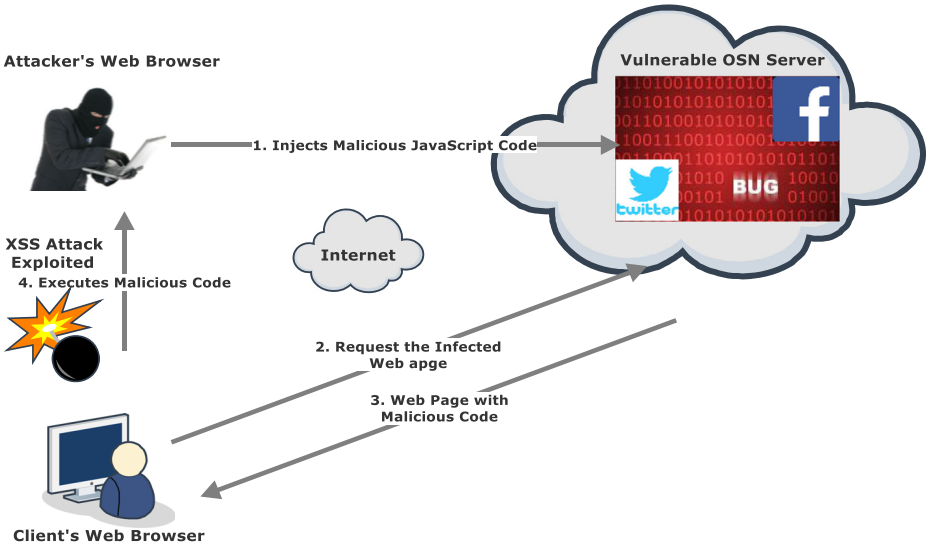

Fig. 1 Injection of XSS worm on the vulnerable cloud-based OSN web server

XSS worm comes in four different flavors: Non-Persistent [11], Persistent [11], DOM-Injection [11] and Mutation-based XSS attacks [11]. Although, the key goal of all these four different categories of XSS worms is to steal the sensitive credentials such as transaction passwords, credit card numbers, etc. of the online user.

However, the technique of exploitation of such worms is different on different platforms of OSN. According to recent survey done by CVE [7] (as shown in the Fig. 2), nearly 65 % of OSN-based web applications are still vulnerable to top most vulnerability i.e. XSS.

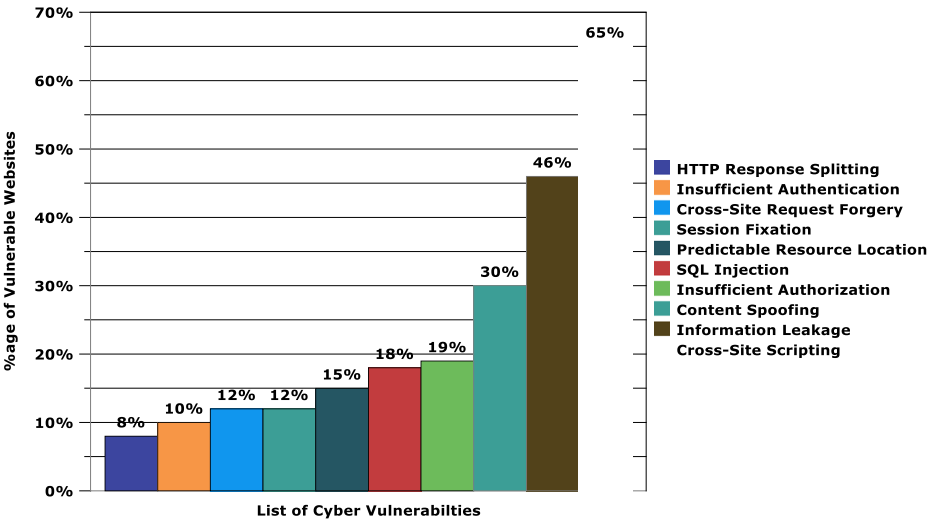

Fig. 2 Percentage of vulnerability of different threats

1.2 Related work

This sub-section discusses the related existing XSS defensive methodologies that are somewhat similar to our work. Numerous client-side and server-side XSS defensive methodologies have been proposed by the web security researchers for detecting and mitigating the effect of XSS worms from the real world web applications. Pelizzi et al. [24] proposed an XSS filter, XSSFilt, which could discover non-persistent XSS vulnerabilities. This filter identifies and thwarts portions of address URL from giving an appearance in web page. This filter could also discover partial script insertions. XSS Auditor [3] is a filter that realizes equally extraordinary performance as well as high accuracy via jamming scripts following the HTML parsing and prior to execution. The filter can simply spot the components of the response which are considered as a script. It also scans the Document Object Model (DOM) tree generated by the HTML parser for the clear interpretation of the semantics of the bytes.

Saxena et al. [27] proposed a technique known as ScriptGard that is a complementary technique that presumes the collection of accurate sanitizers and inject them to match the parsing context of web browser. Path Cutter [6] 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. The authors partition the web application into several views, then Path-Cutter separates these views on the client side by simply permitting those HTTP requests approaching from a view with the authorized privileges. Hooimeijer et al. [16] proposes BEK, a language platform for writing sanitizers which facilitates organized reasoning with reference to their exactness. Saner [2] is a tool that combines static and dynamic analysis practices to detect the defective sanitization techniques that can be evaded by a malicious attacker. The main objective of this tool is to analyze the working of sanitization procedures to report the web application vulnerabilities like XSS attack. Parameshwaran et al. [23] designed is a robust transformation technique for JavaScript web applications. It efficiently evaluates and identifies DOM-based XSS susceptibilities in web applications. It rewrites JavaScript of the requested website to achieve character-precise taint tracing. Gupta et al. [15] presents a XSS vulnerabilities detection tool, XSSDM, which utilizes context sensitive approach with the currently existing static taint analysis method and pattern matching procedure. Stock et al. [28] proposed a technique to examine client-side XSS susceptibility of numerous malicious web sites to detect features of complex JavaScript code. According to the identified features, it outlines a set of metrics to find out the amplitude of the complexity. The engine then collects all the identified vulnerable data flows that happen during the processing of the web page and forwards it to a central database for storage. Once for all URLs under examination have been accumulated, then it is post-processed and can then be evaluated to find the XSS attack. Xiao et al. [32] presents a dynamic information flow tracing technique built on code rewrite mechanisms to stop sensitive information disclosures. It integrates taint and information flow analysis, by modifying the JavaScript code to confirm the integrity and confidentiality of sensitive data

1.3 Existing research gaps

Effective sanitization of XSS worms is considered to be the first and foremost solution for alleviating the effect of XSS worms from the injection points of multimedia-based web applications. However, most of these discussed recent XSS defensive methodologies sanitize the untrusted variables of JavaScript code without finding out their context. The sanitization of

such malicious variables prior to determining their different contexts makes the way for the attacker to bypass such effective sanitization routines. However, the evaluation of XSS attack detection capability of existing XSS defensive frameworks was tested on different web applications by referring only the XSS cheat sheet [26]. In addition, most of these existing works do not focus on the method of sanitizing the latest HTML5 XSS attack vectors. Numerous XSS attack vector repositories are available on the World Wide Web (WWW). The main issue with these existing techniques is that they cannot solve the problem of isolating untrusted data from trusted data. High rate of false positives and false negatives are observed in the existing defensive frameworks of XSS. Moreover, existing frameworks are very difficult to integrate in the settings of cloud platforms [5, 18]. On the other hand, the performance analysis of existing work was rarely tested on the contemporary platforms of OSN.

1.4 Contributions of our work

Based on the existing performance issues in the XSS defensive frameworks, this article presents a cloud-based framework, XSS-Secure that discovers and alleviates the propagation of XSS worms from the multimedia-based OSN web applications. XSS-Secure operate in dual mode: Training and Detection mode. The former mode sanitizes the extracted untrusted variables/links of JavaScript code and stores this sanitized code in the OSN web server for further reference in the detection mode. The later mode compares the sanitized HTTP response (HRES) on the cloud-based OSN web server with the sanitized HRES obtained during the training mode. Any discrepancy observed in the HRES messages will point towards the injection of malicious JavaScript code on the OSN server deployed in the virtual machine of cloud platform. In addition, this mode will find out the different possible contexts of such untrusted/malicious variables and performs the sanitization on them in a context-aware manner. Finally, context-aware sanitized HRES is transmitted to the web browser of OSN user.

We have developed a prototype of XSS-Secure in Java development framework and install all its components of detection mode in different virtual machines of cloud platforms. Evaluation and testing of our work was carried out on the tested platforms of non-OSN as well as OSN-based multimedia web applications. Experimental results revealed that our framework detects the injection of XSS worms with high precision rate, acceptable rate of false negatives and false positives. In addition, the automated procedure of sanitization of untrusted variables of JavaScript code was executed with acceptable performance overhead.

The rest of the paper is organized as follows: In section 2, we describe some of the challenges and issues in existing sanitization-based XSS defensive solutions. In section 3, we introduce our XSS-Secure design in detail. Implementation, experimental evaluation and performance analysis of our work are discussed in section 4. Finally, section 5 concludes the our work and discusses further scope of work.

2 Existing challenges and issues in sanitization-based XSS defensive solutions

This section discusses some of the challenges and research gaps in the existing XSS defensive state-of-art techniques. The sanitization is considered to be the foremost solution for XSS attacks. Here, we will present and explain the challenges inherent in XSS sanitization.

2.1 Dynamic code evaluation

Normally, the web browsers evaluate the source code of web applications in dynamic oriented manner. Therefore, the length of the path navigated by web browser is large enough during parsing of strings of JavaScript. Figure 3 highlights the following JavaScript code snippet:

It is clearly reflected from the code snippet that the untrusted content is recursively called by the function of JavaScript. It is quite challenging for the web browser to determine the series of contexts traversed on the web server due to the evaluation of code in a dynamic manner. However, client-side sanitization is required in such circumstances for completely mitigating the effect of XSS attacks. In addition to this, avoiding sanitizing such dynamic evaluation produces DOM injection category of XSS vulnerabilities. Such category of vulnerabilities occurs due to the wrong interpretation of the behavior of web application by the application frameworks.

2.2 Extent of automated sanitization support

Nowadays, numerous HTML5-based web applications provide the automated support of sanitization, wherever it is required in the possible user injection points of web applications. Weinberger et al. [30] analyzed numerous web application frameworks for auto-sanitization support and the author found that very few existing frameworks support auto-sanitization. Although, those who support, they do not apply diverse sanitizers according to the context in which data flows. This implies that existing frameworks suffer from context-insensitive sanitization. Table 2 highlights the details of auto-sanitization support in different frameworks.

It is clearly reflected from the Table 2, that the existing platforms simply performs the sanitization on the malicious code of JavaScript without determining the context of such variables. The safe interpretation of HRES message on the web browser is not possible by performing simply the sanitization of untrusted variables of JavaScript code without knowing their context. Based on this research gap in the existing sanitization-based XSS defensive solutions, this article presents an OSN-based cloud framework, XSS-Secure. Prior to the execution of automated process of sanitization, our framework finds out the possible different context of untrusted variables of JavaScript code and accordingly performs the sanitization on them. The next section discusses our work in detail.

3 Proposed framework: XSS-Secure

This section discusses our proposed cloud-based framework XSS-Secure, which explores for the untrusted variables of JavaScript code in OSN server and sanitizes such variables in a context-aware manner. The novelty of our framework lies in the fact that it not only detects the injection of XSS worms on the OSN server but also determines the context of such worms prior to the execution of auto-context sanitization procedure on such worms.

Fig. 3 Malicious JS code snippet

```
function abc (malicious data)
{
    document.write("<input onmouseover=' abc (" + malicious data + ")' >");
}
```

Table 2 Summary of auto-sanitization support in existing frameworks

Platform	ECSS	ASHW	SIP	Plugin utilized
Python	✗	✓	Template Handling	Django
C#	✗		Request function	ASP.NET Request Validation
C++	✓		Template Handling	Ctemplate
PHP	✗		Request function	Code igniter
Language-neutral	✓		Template handling	ClearSilver
Ruby	✗		Database injection	xss_terminate Rails plugin
Java	✓		Template handling	GWT SafeHtml

ECAS execute context-sensitive sanitization, *ASHW* automatically sanitizes HTML web applications, *SIP* sanitized injection points

In addition, the evaluation and testing of our framework was not only executed on real world multimedia web applications but also on the contemporary tested platforms of multimedia-based OSN in cloud infrastrustures. The following sub-section discusses the detailed abstract overview of XSS-Secure.

3.1 Abstract overview

XSS-Secure is a runtime parsing and context-aware sanitization based framework that scans for the XSS worms on the multimedia-based OSN web applications in cloud platforms. Figure 4 highlights the abstract design view of our proposed cloud-based framework. The OSN server deployed in the virtual machine of cloud platform initially extracts the HTTP request (HREQ) from the users of OSN. The OSN server utilizes the

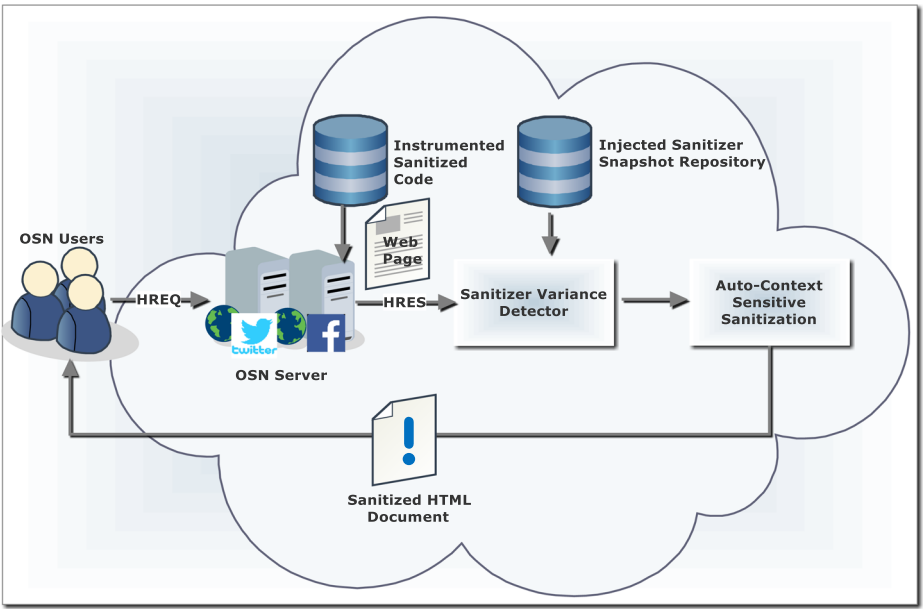

Fig. 4 Abstract design view of cloud-based XSS-secure

instrumented sanitized code repository for sanitizing the untrusted variables of JavaScript and finally generates an HTTP response (HRES) in the form of a web page. The sanitizer variance detector compares the value of untrusted variables that are sanitized on the OSN server with the sanitizers stored at the injected sanitizer snapshot repository. Any variation observed in the injected values of sanitizers indicates the injection of XSS worms on the OSN server. The sanitizer variance detector not only detects such variation but also recognizes the injection of untrusted JavaScript code on the OSN server. The detection of such untrusted JavaScript code will be transmitted to the auto-context sensitive sanitization component that determines the context of such variables. Based on the determined context of such variables, this component will perform the context-sensitive sanitization on such untrusted variables and finally generates and transmits a sanitized safe HTML document to the OSN user.

Detailed design overview XSS-Secure executes in two modes: training and detection mode. Figure 5 highlights the detailed overview of XSS-Secure. The key goal of the training mode is to generate the safe sanitized JavaScript code embedded in the templates of web pages. The training mode initially extracts the series of HREQ and accordingly generates a set of session states. The series of HRES messages are generated on the OSN server corresponding to each extracted session state. The dynamic parser component will perform the dynamic parsing of HRES messages for identifying the different context of embedded variables. The template generator will produce the resultant set of templates of web pages corresponding to all parsed HRES messages. The hidden injection point generator will crawl all the extracted templates of web pages for locating the hidden injection points. In addition, the JavaScript link extractor will explore for the untrusted variables of JavaScript and links injected in the extracted hidden injection points. Context-aware sanitizers will be applied on such malicious variables and links by utilizing the sanitizer library. Finally, this instrumented sanitized JavaScript code of different templates is transmitted to the sanitizer snapshot repository and OSN server for further instrumentation during the detection mode.

On the other hand, the key goal of the detection mode is to detect the variation in the injected sanitizers during the offline mode. In addition, it also detects the injection of malicious variables and links of JavaScript code. In order to avoid this, the detection mode determines the context of such untrusted variables of JavaScript code and performs the context-sensitive sanitization on such variables for their safe interpretation on the web browser. Initially, this mode extracts the online series of HREQs on the cloud platform. The instrumented OSN server generates a HRES message comprising the embedded sanitized JavaScript code produced during the offline mode. This sanitized HRES message will be transmitted to the sanitizer variance for detecting the variation in the stored sanitized variables/links at the sanitizer snapshot repository and the sanitized variables/links embedded in the HRES message. Any variation observed in the sanitized HRES message will be identified as an XSS worm. In addition, the sanitizer variance also detects the injection of malicious JavaScript code on the OSN server. The variable context finder determines the context of such variables and accordingly performs the context-sensitive sanitization on such untrusted variables for their safe interpretation on the web browser of OSN user. The working flow of both the offline and online modes of our proposed framework are illustrated in Fig. 6.

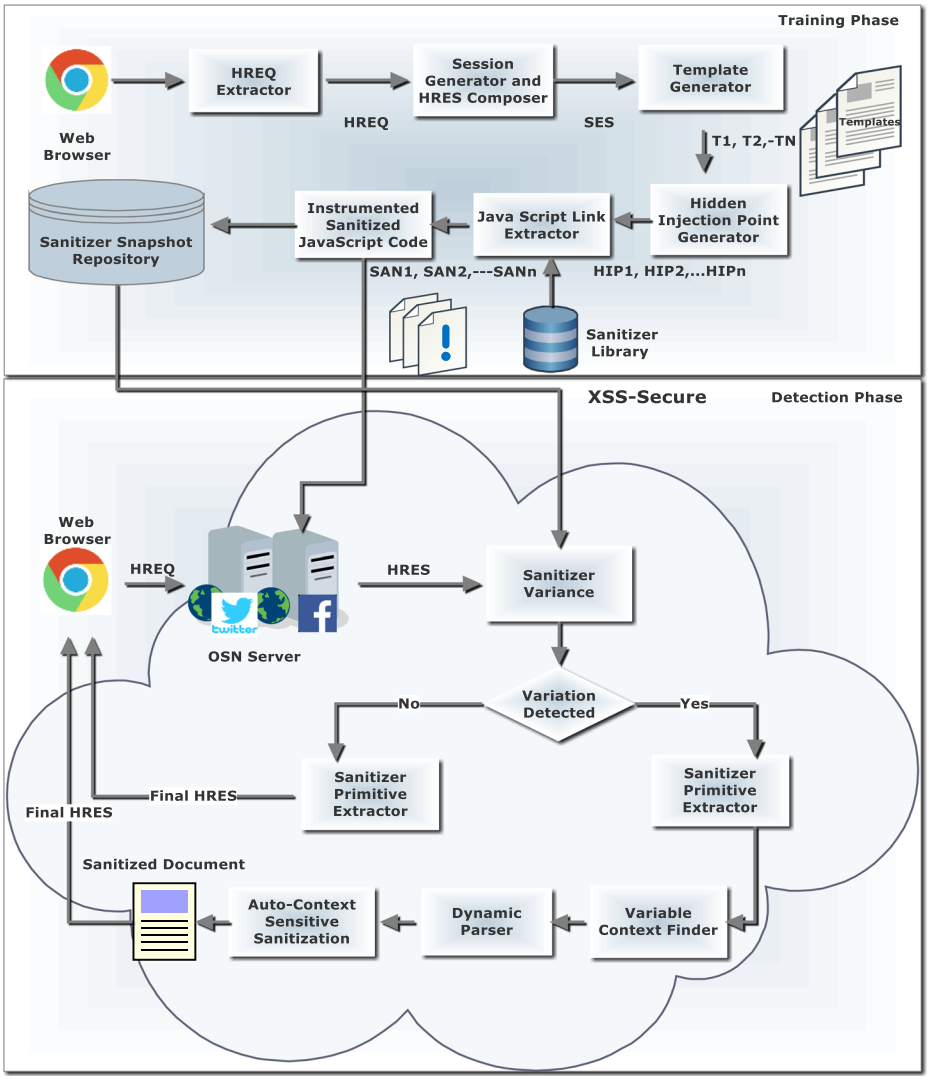

Fig. 5 Detailed design view of XSS-Secure

3.2 Key components of XSS-Secure

Note that, the components of detection mode are deployed in the virtual machines of cloud platforms. However, the components of training mode are deployed on the server-side. This sub-section discusses the detailed working of some of the key components utilized in the training and detection modes of proposed cloud-based framework.

3.2.1 Session generator and HTTP response (HRES) composer

The key goal of this module is to generate the values of session ids (SES_1, SES_2, ——— SES_N) corresponding to each extracted series of HREQ messages during the offline phase.

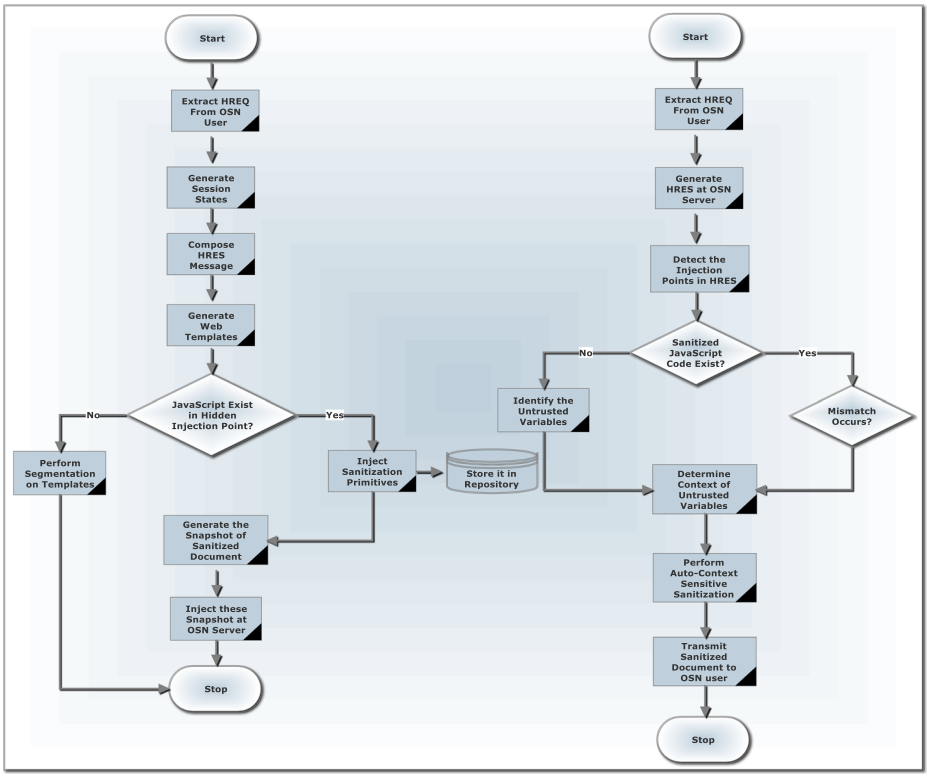

Fig. 6 Flowchart of operation of XSS-secure 6 (**a**) Offline mode 6 (**b**) Online mode

Such session ids will be utilized for the generation of HRES messages ($HRES_1$, $HRES_2$, ————, $HRES_N$).

3.2.2 Template generator

This component will produce the set of templates (T_1, T_2, ——— T_N) of HTML web pages. The series of such templates will be utilized for detecting the hidden injection points. In addition, we have also explored for the untrusted variables of JavaScript code in such injection points. The execution of such payloads of JavaScript code will result in compromise of credentials of the online user like cookies, credit card numbers, etc. Figure 7 highlights the proposed algorithm of template generation at the training phase.

Here, for each extracted HREQ, generate the session id (S_I) and store all the sequence of session ids in the session log file (S). Hereafter, for every generated session id (S_I), generate the list of templates for corresponding HREQs. Now, in order to differentiate between the extracted session ids and the associated templates, this algorithm generates a unique pair of (HREQ, HRES) for HTTP request and HTTP responses. Finally, we generated some sort of axioms (A) that will uniquely identify the templates for the associated session ids by utilizing the (<key, value>) pair.

Algorithm: Session and Web Template Generator

Input: Set of transmitted HREQs ($HREQ_1$, $HREQ_2$, ---- $HREQ_N$)
Output: Retrieved Set of Web Templates (T_1, T_2, -------, T_N)

```
Start
S ⇐ ϕ ;                    /* list to store generated session*/
HREQ ⇐ List of HTTP requests;
For each HTTP request HI ∈ HREQ
      Generate session SI;
      S ⇐ SI ∪ S;
End For each
For each SI ∈ S
      Generate template TI for HI;
      <key, pair> ⇐ (HREQ, HRES);
      A ⇐ Axiom-generator(<key, value>);
      Return A
End For each
End
```

Fig. 7 Proposed algorithm of session and web template generator

3.2.3 Hidden injection point generator

Once the set of templates are retrieved based on the associated session ids of HREQ and HRES messages, the next step is to locate the hidden injection points from such web templates. Figure 8 highlights the proposed algorithm of hidden injection point locator.

Here, we utilize the URL of blacklisted web pages as an input to this algorithm. We have saved all these URLs in the URL log file, i.e., URL_REP. The algorithm utilizes the capabilities of web crawler engine to explore all the hidden user-injection points (HIP) of such blacklisted URLs. The web crawler explores the blacklisted URLs for the maximum time

Algorithm: Hidden Injection Point Locator for Web Templates

Input: Extracted URL of blacklisted web pages (W_1, W_2, W_3, --------- W_N)
Output: Series of Extracted Hidden Injection Points (HIP_1, HIP_2, HIP_3, -------- HIP_N)

```
Threshold ( φ ) ⇒ 0
Start
URL_REP ⇐ ϕ ;
MC_REP ⇐ Malicious Context Repository ;
HIP_REP ⇐ ϕ ;
W_REP ⇐ ϕ ;
Initialize: = URL_REP ⇐ WI;
φ ← ranGen();
UI ←0;
HIPI ←0;
For J ←1 to φ
      For Each WI ∈ URL_REP
            W_REP ⇐ WI ;
            Explore Malicious-context(WI) ⇔ MC_REP;
            UI ← URL(WI);
            HIPI ← HIP(WI);
            URL_REP ← UI ∪ URL_REP;
            HIP_REP ←HIPI ∪ HIP_REP;
      End For Each
End For
Return HIP_REP
End
```

Fig. 8 Proposed algorithm of hidden injection point locator

produced by the function ranGen(). Hereafter, in all the iterations up to the randomly selected value of threshold; we extract malicious injection points in W_REP corresponding to each blacklisted website W_I. Now, simply extract the links of these websites as blacklisted URLs in URL repository (URL_REP). Finally, merge all these injection points in HIP_I with hidden injection point repository (HIP_REP).

3.2.4 JavaScript link extractor

After the successful identification of hidden injection points from the blacklisted URL addresses of extracted templates, the next step is to extract the JavaScript links from such injection points by using JavaScript link extractor component. Figure 9 illustrates the detailed proposed algorithm of JavaScript link extraction. The algorithm utilizes the extracted hidden injection points and web links for extracting the JavaScript worms. In addition, the algorithm has also extracted the XSS worms from the XSS attack vector repositories (as listed in the Table 6) and stored this collection in the repository.

Here, for each extracted HTTP request as E_HREQ, simply extract all the parameters in $Param_I$ and merge all these parameters with Param_Rep. Hereafter, the algorithm will check for the URLs containing parameters in Param_Rep as URL_I. and if URL_I exist, transmit the AJAX request to the remote server. In addition, extract the JS worms in JL_REP present at all the hidden injection points in HIP_Rep.

3.2.5 Sanitizer variance

The key goal of this module of detection mode is to detect the XSS attack by perceiving the variation in the value of sanitizers embedded in the HRES message and the value of sanitizers stored in the sanitizer snapshot repository. Any variation observed points

Fig. 9 Proposed algorithm of JavaScript link extractor

Algorithm: JavaScript (JS) Link Extractor

Input $\Rightarrow$ Series of HREQ ($HREQ_1$, $HREQ_2$, $HREQ_3$, ----- $HREQ_N$)

Output $\Rightarrow$ **Series of JavaScript Links** (JL_1, JL_2, JL_3,----------JL_N)

Start

E_HREQ $\Leftarrow$ Extracted HTTP request;

JL_REP $\Leftarrow \phi$;

HIP_Rep $\Leftarrow$ Set of extracted hidden injection points;

Param_Rep $\Leftarrow \phi$;

$URL_I \Leftarrow 0$;

$Param_I \Leftarrow 0$;

For Each HREQ $\in$ E_HREQ

  $Param_I \Leftarrow$ Param(HREQ);

  Param_Rep $\Leftarrow$ $Param_I \cup$ Param_Rep;

End For Each

For Each $Param_I \in$ Param_Rep

  $URL_I \Leftarrow$ Verify for URL $\in Param_I$;

  If ($\exists$ URL_I)

    Transmit asynchronous JS request to the isolated server;

    JL_REP $\Leftarrow$ (Extract-JL($Param_I$) $\mapsto$ HIP_Rep);

  End if

End For Each

End

towards the injection of XSS worms. Figure 10 highlights the proposed algorithm for detecting the XSS worms by this module. Here, the list of parameter values embedded in the HRES message are extracted and saved into the variable P_log. Now, for each extracted parameter (P_I), check the existence of Uniform Resource Identifier (URI) links. If there exist URI links, transmit the Asynchronous JavaScript XML (AJAX) request to the remote server for the extraction of JavaScript links and saved such links in variable 'A'. Otherwise, extract the static tags (STA) embedded in the web pages (W_I). Finally, compare the values of set of extracted JavaScript links and static tags embedded in the web pages. Any similarity observed in both these set of tags will be marked as the injection of XSS worm. The similar untrusted JavaScript code found will be sanitized in a context-sensitive manner by following the algorithm explained in the Fig. 12.

3.2.6 Variable context finder

The goal of this component is to identify the context for each type of untrusted variable of JavaScript corresponding to each untrusted input. It will represent the context in which untrusted JavaScript input is embedded and sanitizers are selected accordingly. Table 3 highlights some of the contexts of the components of JavaScript code.

Fig. 10 Proposed algorithm of sanitizer variance

Algorithm: XSS worm detection
Input: Set of extracted HTTP responses ($HRES_1$, $HRES_2$, ---------- $HRES_N$)
Output: Detection of XSS worm in the web page.

```
Start
HRES ⇐ list of HTTP requests;
A ⇐ ϕ ;                    /* list to store extracted JavaScript nodes */
P_log ⇐ ϕ ;                /* list to hold parameters values*/
STA ⇐ ϕ ;                  /* list to store static tags*/
B ⇐ ϕ ;                    /* list to store scripts nodes*/
For Each HI ∈ HRES
    PI ⇐ Parameter(HI);
    P_log ⇐ PI ∪ P_log;
End For Each
For Each PI ∈ P_log
    UI ⇐ Check for URI ∈ PI;
    If ( ∃ UI)
        Send AJAX request to server;
        A ⇐ (Extract-JS-links(PI);
    Else
        WI ⇐ Extract-webpage(UI);
        DI ⇐ HTML-parser(WI);
        STA ⇐ Extract-static-tags(DI);
        If (( ∃ x ∈ STA): DYNS) then
            SI ⇐ Sanitizer(x);
        End if
        B ⇐ Script-nodes(DI);
        If(( ∃ x ∈ A)==( ∃ y ∈ B)) then
            Return XSS detected;
        Else
            Return Safe document;
        End if
    End if
End For Each
End
```

Table 3 Different categories of context of JavaScript code

Sr. no.	Components	Context
1.	JavaScript	REGEX, String
2.	URL	Query, Start, General
3.	HTML	RCDATA, ATTRIBNAME
4.	CSS	QUANT, PROPNAME, Quoted URL
5.	HTMLATTRIB	Quoted, Unquoted

3.2.7 Dynamic parser

This component isolates the visual, linking elements and scripting elements and passed them to the CSS parser, URI parser and JavaScript parser respectively. These parsers process the content received. In addition to this, they also dynamically determine the context of the untrusted variables of JavaScript to which a dynamic type qualifier is attached. Its main goal is to construct a parse tree i.e. Document Object Model (DOM). During parsing, executable script nodes are determined and nodes are created for them in the parse tree.

Figure 11 describes the algorithm used to execute the module of dynamic parsing. It provides the step wise working process of this offline module. HTML parser generates document SW corresponds to each HRES. Then, for each static type qualifier Q, applies the sanitizers on the variables injected by the CASG phase and displays the sanitized document to the user. For all type promotions operations as ($\Gamma \mapsto UV_I$: DYN_S), we parse the document to dynamically identify the context of the untrusted input in which it can be safely interpreted. SW_J stores the result of the JS_Parser. SW_C holds the result of CSS_Parser and D_U stores the result of URI_Parser. After applying parsers, we execute the sanitizer routines on the resultant typed document. San (UV_I, f) function will executes sanitizers' primitive f on UV_I according to the matching context indicated by the attached type qualifier. Finally, the sanitized HTML document is displayed to the user at the browser window.

3.2.8 Auto context-sensitive sanitization

It is a process of validating the untrusted user input to ensure that they are in correct format as perceived by the Web application. Auto Context-sensitive sanitization applies sanitizer on each untrusted variable in an automated manner (i.e. dynamic content like JavaScript) according to the context in which they are used. There may be different contexts present in an HTML document like element tag, attribute value, style sheet, script, anchors, href, etc. These all contexts may be used by the attacker to launch the XSS attack. Figure 12 illustrates the proposed algorithm of sanitizing the untrusted variables of JavaScript code. Here, for each extracted JS link $JL_I \in JL\_REP$, the proposed algorithm recognizes untrusted variables ($Untrust_{VAR}$) and determined its context as CF_I. Furthermore, extract the sanitizer primitive (SAN_I) from externally available sanitizers library (S_LOG) that matches the context (CF_I). Now, simply apply this sanitizer SAN_I on JL_I and merge this result with the sanitizer library for the effective results. Finally, replace the untrusted variable ($Untrust_{VAR}$) with the sanitized variable. Now finally, sanitized HRES is returned to the cloud user.

In order to prevent XSS attacks, web application requires confirming that every variable outputs associated with a web page are properly encoded/encrypted before being reverted

Algorithm: Dynamic Parsing

Input: Series of extracted HTTP responses ($HRES_1$, $HRES_2$, … $HRES_N$)
Output: Series of Sanitized HTTP response ($SHRES_1$, $SHRES_2$, … $SHRES_N$)

Start
SW $\Leftarrow$ Null
For Each HTTP response as HRES
SW $\Leftarrow$ HTMLParse (HRES)
For Each UV_I
If (UV_I : $STA_{C' \rightarrow C''}$) **then**
Inject CSS at UV_I
Return sanitized HTML web page SHRES
End If
End For Each
For Each type qualifier variable (UV_I) $UV_I \in$ (HTML element/attribute $\cap$ $\Gamma \mapsto {}_U V_I : DYN_S$)
San(UV_I, f) f$\in$ S (S$\in$ 2^{C*C})
End For Each
$SW_J \leftarrow$ JS_Parse (SW)
For Each type qualifier variable (UV_I) $UV_I \in$ (JavaScript context $\cap$ $\Gamma \mapsto {}_U V_I : DYN_S$)
San(UV_I, f) f$\in$ S (S$\in$ 2^{C*C})
End For Each
$SW_C \leftarrow$ CSS_Parse(SW_I)
For Each type qualifier variable (UV_I) $UV_I \in$ (CSS context $\cap$ $\Gamma \mapsto {}_U V_I : DYN_S$)
San(UV_I, f) f$\in$ S (S$\in$ 2^{C*C})
End For Each
$D_U \leftarrow$ URI_Parse (SW_C)
For Each type qualifier variable (UV_I) $UV_I \in$ (URL context $\cap$ $\Gamma \mapsto {}_U V_I : DYN_S$)
San(UV_I, f) f$\in$ S (S$\in$ 2^{C*C})
End For Each
Inject CSS at each UV_I
Return sanitized HTML web page SHRES
End For Each
End

Fig. 11 Proposed algorithm of dynamic parsing

towards the web browser. Thwarting XSS attack vectors implies to replace all possible exceptional characters utilized in the exploitation of such attacks. Our technique has escaped

Fig. 12 Proposed algorithm of sanitization of extracted JavaScript links

Algorithm: Sanitization of untrusted variables of JavaScript code.

Input: Series of extracted JavaScript (JS) Links (JL_1, JL_2, JL_3,---------- JL_N)
Output: Sanitized HTML document

Start
S_LOG $\Leftarrow$ Whitelist of Context-Sensitive Sanitizers;
JL_REP $\Leftarrow$ Whitelist of external JavaScript Links;
$Untrust_{VAR} \Leftarrow$ NULL
$San_{VAR} \Leftarrow \phi$;
For Each Extracted JS Link $JL_I \in$ JL_REP
$Untrust_{VAR} \Leftarrow Untrust_{VAR}(JL_I)$;
$CF_I \Leftarrow$ Context_Funder ($Untrust_{VAR}$);
$SAN_I \Leftarrow$ (San $\in$ S_Log) $\cap$ (San MATCHES CF_I);
$San_{VAR} \Leftarrow SAN_I(JL_I)$;
S_LOG $\Leftarrow San_{VAR} \cup$ S_LOG;

$Untrust_{VAR} \Leftrightarrow SAN_I(Untrust_{VAR})$;
End For Each
Return sanitized document;
End

the malicious characters through utilizing the &# arrangement followed through its code of character. Table 4 highlights some of the list of escape codes.

In our work, we have described the technique of sanitizing the different categories of JavaScript code. We have developed some manual functions that are utilized for sanitizing the malicious code of JavaScript.

3.2.9 Sanitization of doubtful keywords

A doubtful keyword falls in the category of identified malicious data for blocking illegitimate data from execution. Our technique has utilized a fixed template of sanitization which will sanitize the user input in a precise manner. If any user-injected string matches the template of sanitization, then it is recognized as a XSS worm and would be further substituted with resultant keywords.

Usually, document.write, parentNode, <embed, document.cookie, <script etc. are considered to be some of the insecure or doubtful keywords. Our technique has statically generated a compiled list of blacklisted keywords and blocked those JS attack vectors whose keywords falls in those blacklisted category. The details of this are as shown in the Fig. 13.

3.2.10 Sanitization of character escaping

In order to alleviate XSS worms, web application requires confirming that every variable outputs associated with a web page are properly encoded/encrypted before being reverted towards the web browser. Thwarting XSS attack vectors implies to replace all possible exceptional characters utilized in the exploitation of such attacks. Our technique has escaped the malicious characters through utilizing the &# arrangement followed through its code of character. Table 5 highlights some of the list of escape codes.

Suppose malicious attacker inserts < script > alert("XSS")</script > inside the variable area of victim's web application, then the special character like '<', '>' would be replaced with &#60 and &#62. Now the web browser will show this script in the form of a portion of a web page, however the web browser could not run the script. Figure 14 illustrates the technique related to filtering character escaping. We have incorporated the feature of list of common

Table 4 List of escape codes of HTML character symbols

S. no.	HTML character symbols	Alternative representation
1.	{	%7b
2.	}	%7d
3.	+	&#43
4.	-	&#45
5.	%	&#37
6.	#	&#35
7.	@	%40
8.	;	&#59
9.	~	%7e
10.	"	&#34

Fig. 13 Function for sanitizing insecure keywords

```
public function sanitize_blacklisted($str)
{
   strtolower($str);
   $blacklisted = array(
            'document.cookie'     => '';
            'document.write'      => '';
            '<!—'                 => '&lt;!—';
            '-->'                 => '--&gt;';
            '<! [CDATA['          => '&lt; ! [CDATA[',
            '<comment>'           => '&lt;comment&gt;';
            '.parentNode          => '',
            '.innerHTML'          => '',
            'window.location'     => '',
            '-moz-binding'        => '',
            '<embed'              => '',
            '<applet'             => '',
            '<object'             => '',
            '<script'             => '',
   );
$str = str_ireplace(array_keys($blacklisted), $blacklisted, $str);
return $str;
}
```

escape codes in our technique and thus, XSS-Secure can stop the web browsers of OSN users from executing such scripts.

3.3 Additional security analysis of XSS-Secure

The existing XSS defensive frameworks utilize only the XSS cheat sheet repository [26] for accessing the XSS attack vectors. However, this repository of XSS worms is not sufficient for evaluating the detection and mitigation capability of robust XSS defensive framework. Keeping in this view, we manually verified the performance of XSS-Secure against five open source available XSS attack repositories, which includes the list of old and new XSS attack vectors. Such repositories also include current HTML5 XSS attack vectors. Very few XSS attack vectors were able to bypass our cloud-based design. Table 6 highlights the details of five open source available XSS attack repositories.

In addition to this, XSS-Secure verify the files saved at the web servers for a fixed range of HTML tags in order to defend against XSS attacks. We have chosen a subset of HTML tags (as highlighted in Table 7) that can run the script in an automated fashion. We have selected HTML tags that: could comprise codes, copy as well as run the codes in an automated manner, comprise automatic forwards, and include the set of attributes that could include codes.

Table 5 List of escape codes

Display	Hexadecimal code	Numerical code
“	"	"
#	#	#
&	&	&
‘	'	'
(	(	(
)	)	)
/	/	/
;	;	;
<	<	<
>	>	>

Fig. 14 Function for sanitizing character escaping

```
Public function filter_char_esc($string)
{
    $str = str_replace
        (
            array('<', '>', '"', '"', ')', '(' ),
            array('&#x3C;', '&#x3E;', '&#x27;', '&#x22;', '&#x29;', '&#x28;')
            $string
        );
    $str = str_ireplace( '%3Cscript', ' ', $str );
    return $str;
}
```

The symbol # in Table 7 indicates any character, which can be probable value of attribute that is not considered to be appropriate for any malicious action. The script HTML element could encompass dangerous code as well as the link and object tags could automatically copy and run dangerous code. HTML attributes like object, frameset, and body elements could include event handlers (as highlighted in Table 8) that again could run dangerous code in an automated manner. Hence, our cloud-based framework considers such HTML attributes as well as elements as possibly malicious.

In addition, our framework have also carefully access the Document Object Model (DOM) properties of the JavaScript document. The objects retrieved by the DOM tree could be any attribute or keyword that can also result in stealing of cookie file and history related to that web page, that can be further useful for tracking the surfing behavior of cloud user. In order to avoid this, we have maintained a blacklist of DOM properties that are used for the exploitation of XSS worms. Table 9 highlights the list of such DOM properties.

3.4 Merits of XSS-Secure

The novelty of XSS-Secure lies in the fact that it not only refers the XSS cheat sheet [26] for accessing the XSS attack vectors but also access the untrusted JavaScript attack vectors in the other four XSS attack repositories listed in the Table 6. This increases the robustness of our approach. The sanitizer variance component not only detects the variation in the sanitized variables of untrusted JavaScript code but also detects the injection of JavaScript worms on the OSN server in detection mode. The recent XSS defensive methodologies clearly sanitize the variables of JavaScript code without determining the context of such variables. However, prior to sanitization, XSS-Secure determines the context background of such malicious variables and then performs the sanitization on such variables accordingly in a context-aware manner.

Table 6 List of XSS attack vector repositories

Sr. no.	XSS attack vector repositories	Reference
1.	XSS Filter Evasion Cheat Sheet	https://www.owasp.org/index.php/XSS Filter Evasion Cheat Sheet
2.	HTML5 Security Cheat Sheet	http://html5sec.org/
3.	523 XSS vectors available	http://xss2.technomancie.net/vectors/
4.	Technical Attack Sheet for Cross Site Penetration Tests	http://www.vulnerability-lab.com/resources/documents/531.txt.
5.	@XSS Vector Twitter Account	https://twitter.com/XSSVector

Table 7 Category of possible malicious HTML attributes and their malicious restricted syntax

Sr. no.	HTML element	Description	Attack example
1.	script	Script elements	<script # > #</script>
2.	applet	Java Applet	<applet # code src=
3.	frameset	Window subdivision	<frame set # [events] =
4.	img	src	
5.	input	src	<input type = "image" src = "attack_malicious URL" alt = "Submit"/>
6.	table	HTML Table	<table # background=
7.	body	background	<body background: url('attack_malicious URL')>
8.	link	href	<link type = "text/css" href = "attack_malicious URL" />
9.	object	Generic embedded object	<object # [events] = object < object # > <param <object # data = <object # codebase=

On the other hand, the existing work cannot properly evaluate and tested the capabilities of their framework on the contemporary platforms of OSN. Most of the existing techniques were tested on the platforms of non-OSN web applications. However, the detection and alleviation capability of XSS-worm of our cloud-based framework was tested on tested platforms of non-OSN as well as OSN web sites. This escalates the scalability of our work.

4 Implementation and experimental evaluation

This section discusses the implementation and experimental evaluation outcomes of our proposed cloud-based framework.

4.1 Implementation

We have developed a prototype of XSS-Secure in Java and integrate this cloud-based framework in the virtual machines of cloud servers. We have implemented the design of our proposed framework in two modes: training and detection mode. All the components of training mode are implemented in Java and integrated their functionality on the OSN web server for further instrumentation in the detection mode. The capability of some of the

Table 8 List of onload events

Sr. no.	HTML attributes	List of events
1.	Window	onbeforeprint, onstorage, onhashchange, onpageshow, onload
2.	Mouse	onclick, ondrop, ondrag, onscroll, ondragend, ondragover, onwheel
3.	Form	oninvalid, onreset, onformchange, onforminput, onblur, onfocus
4.	Media	onsuspend, onloadstart, onprogress, ontimeupdate, onloadedmetadata.
5.	Misc	Onerror, ontoggle, onshow.
6.	Keyboard	onkeyup, onkeydown, onkeypress
7.	Clipboard	oncut, oncopy, onpaste

Table 9 Malicious properties of DOM and resultant attack exploitation description

Sr. no.	DOM property	Attack explanation
1.	document.getElementByid	This can be utilized to adjust the values of attributes and tags in the web page.
2.	document.writeln	This property can be utilized to modify the content of web page.
3.	History.forward	This property can be accessed for traversing and accessing the history of web browser tabs.
4.	Document.cookie	This property is utilized to retrieve the values of cookies of a particular session.
5.	window.location.reload	This property can be utilized to change the position of document.

components of training mode is also utilized in detection mode. In addition to this, all the integrated modules of detection mode are also developed in Java platform and assembled their functionality on the virtual cloud server. The proficiency of such modules is only utilized in detection mode. We have also utilized the VMware Workstation 7 for modifying two virtual desktop systems to integrate the components of training mode and detection mode including one desktop system to be act as OSN web server.

4.2 Experimental outcomes

The experiment background is simulated with the help of a normal desktop system, comprising 1.6 GHz AMD processor, 2 GB DDR RAM and Windows 7 operating system. The XSS attack detection capability of XSS-Secure was evaluated on a tested suite of five open source non-OSN web applications and five OSN-based multimedia web sites deployed in virtual machines of cloud platforms. Tables 10 and 11 highlight the detailed configuration of these web applications.

We estimate the testing, accuracy and the performance of our cloud-based proposed framework. In terms of testing, we have tested our cloud-based XSS defensive mechanism on five open-source real world multimedia web applications (i.e., Wackopicko, Scarf, BlogIt, PhpBBv2 and OsCommerce). We have also tested the XSS worm detection capability of our proposed system on five open source real world multimedia OSN platforms (i.e. Humhub,

Table 10 Configuration of non-OSN platforms of multimedia web applications

Web application	Version	Description	Quantity of files	Lines of code
Wackopicko [29]	2.0	Blogging Board	13	17,884
Scarf	2006-09-20	Conference Management System	31	697
BlogIt [4]	2.0	Blog Application	16	767
PHPBB v2 [25]	2.0.4	Dialogue Forum	12	16,024
OsCommerce [22]	3.0.2	e-Commerce Web Application	19	237,623

Table 11 Configuration of platforms of OSN-based multimedia web applications

OSN Web application	Version	XSS vulnerability	Lines of code
Humhub [17]	0.10.0	CVE-2014-9528	129512
Elgg [9]	1.8.16	CVE-2012-6561	114735
Drupal [8]	7.23	CVE-2012-0826	43835
WordPress [31]	3.6.1	CVE-2013-5738	135540
Joomla [19]	3.2.0	CVE-2013-5738	227351

Elgg, Drupal, Wordpress and Joomla). All the platforms of such web applications have numerous injection points that are vulnerable to XSS attacks. In order to include the capabilities of our cloud-based framework in these ten web applications, developers of these websites have to do less quantity of effort. The motivation to select these two different platforms of multimedia-based web applications is that, we simply need to mark an argument that web sites deployed in the cloud settings can utilize the capabilities of XSS-Secure, irrespective of the input testing. This will aid in alleviating XSS malicious code injection vulnerability concerns and would include supplementary security layer.

The simplicity with which we are capable to combine XSS-Secure in these popular non-OSN and OSN-based multimedia web applications determines the flexible compatibility of our framework. We deployed such web applications on an XAMPP web server with MySQL server as the backend database. We select such applications for accessing the forms to potentially supply modified pages and access the HTML forms. Table 12 highlights the five different categories of the malicious HTML context, including JavaScript Attack Vectors (JSAV), URL Attack Vectors (UAV), HTML Malicious Tags (HMT), CSS Attack Vectors

Table 12 Categories of different context of XSS worms

Context	Pattern example
JSAV	<SCRIPT/SRC = “http://ha.ckers.org/xss.js”></SCRIPT> <<SCRIPT > alert(“XSS”);//<</SCRIPT> <SCRIPT a = “ > ” SRC = “http://ha.ckers.org/xss.js”></SCRIPT> <SCRIPT/XSS SRC = “http://ha.ckers.org/xss.js”></SCRIPT>
UAV	 xxs link <A HREF = “http://ha.ckers.org@google”>XSS</A> <A HREF = “javascript:document.location = ‘http://www.google.com/”>XSS</A> <A HREF = “http://0102.0146.0007.00000223/”>XSS</A>
HMT	<DIV STYLE=“width: expression(alert(‘XSS’));”> <BODY BACKGROUND = “javascript:alert(‘XSS’)”> <OBJECT TYPE = “text/x-scriptlet” DATA = “http://ha.ckers.org/scriptlet.html”></OBJECT> <TABLE BACKGROUND = “javascript:alert(‘XSS’)”>
CSSAV	<LINK REL = “stylesheet” HREF = “http://ha.ckers.org/xss.css”> <STYLE > li {list-style-image: url(“javascript:alert(‘XSS’)”);}</STYLE > <UL > <LI > XSS</br> <STYLE > BODY{-moz-binding:url(“http://ha.ckers.org/xssmoz.xml#xss”)}</STYLE> <STYLE > @import‘http://ha.ckers.org/xss.css’</STYLE>
HMEH	<video onerror = “alert(1)” > <source > </source > </video> <img src = asdf onerror = alert(document.cookie)> <IMG SRC = onmouseover = “alert(‘xss’)”>

(CSSAV) and HTML Malicious Event handler (HMEH). The categories of different context of attack vectors are generated using the five open source XSS attack vector repositories as listed in the Table 6. The observed results of XSS-Secure on two different platforms of web applications deployed in the virtual machines of cloud servers are highlighted in Tables 13 and 14.

Table 13 Observed results of cloud-based XSS-secure on non-OSN based platforms of multimedia web applications

Performance parameters Malicious attack vector categories	# of malicious scripts injected	# of TP	# of FP	# of TN	# of FN
Wackopicko					
JSAV	35	30	1	2	2
UAV	20	16	1	2	1
HMT	15	13	1	0	1
CSSAV	22	20	2	0	0
HMEH	35	32	1	2	0
Total	127	111	6	6	4
Scarf					
JSAV	35	31	2	1	1
UAV	20	16	1	2	1
HMT	15	14	1	0	0
CSSAV	22	19	1	1	1
HMEH	35	30	1	2	2
Total	127	110	6	6	5
BlogIt					
JSAV	35	33	1	1	0
UAV	20	18	1	1	0
HMT	15	12	1	1	1
CSSAV	22	19	1	2	0
HMEH	35	32	1	1	1
Total	127	114	5	6	2
PHPBB v2					
JSAV	35	30	2	2	1
UAV	20	16	1	2	1
HMT	15	12	1	1	1
CSSAV	22	21	0	1	0
HMEH	35	31	2	1	1
Total	127	110	6	7	4
OsCommerce					
JSAV	35	32	1	1	1
UAV	20	17	1	2	0
HMT	15	12	2	1	0
CSSAV	22	19	2	1	0
HMEH	35	31	2	1	1
Total	127	111	8	6	2

Table 14 Observed results of XSS-Secure on OSN-based platforms of multimedia web applications

Performance parameters Attack vectors categories	# of malicious scripts injected	# of TP	# of FP	# of TN	# of FN
Humhub					
JSAV	35	32	1	1	1
UAV	20	18	1	1	0
HMT	15	13	1	0	1
CSSAV	22	20	2	0	0
HMEH	35	33	0	2	0
Total	127	116	5	4	2
Elgg					
JSAV	35	33	2	0	0
UAV	20	18	0	2	0
HMT	15	14	1	0	0
CSSAV	22	18	1	2	1
HMEH	35	32	1	1	1
Total	127	115	5	5	2
Drupal					
JSAV	35	31	3	1	0
UAV	20	17	1	1	1
HMT	15	12	2	1	0
CSSAV	22	19	1	2	0
HMEH	35	33	0	1	1
Total	127	112	7	6	2
Wordpress					
JSAV	35	32	2	1	0
UAV	20	17	0	2	1
HMT	15	13	1	1	0
CSSAV	22	20	1	1	0
HMEH	35	32	2	0	1
Total	127	114	6	5	2
Joomla					
JSAV	35	33	2	0	0
UAV	20	18	0	2	0
HMT	15	13	2	0	0
CSSAV	22	20	2	0	0
HMEH	35	33	0	1	1
Total	127	117	6	3	1

We have injected the five different contexts of XSS worms on the injection points of these web applications. We have analyzed the experimental results of XSS-Secure based on five parameters (# of malicious script injected, # of True Positives (TP), # of False Positives (FP), # of True Negatives (TN) and # of False Negatives (FN)). It can be clearly observed from the Table 13 that the highest numbers of TPs are observed in BlogIt web application. In addition,

the detection rate of TPs is almost 90 %. In case of OSN-based web applications, it can be observed from the Table 14 that the highest of TPs are observed in Joomla.

In addition, the observed rate of false positives and false negatives is acceptable in both the platforms of multimedia-based web applications. We have also calculated the XSS attack payload detection rate of XSS-Secure on both these different platforms of web applications. This is done by dividing the number (#) of TPs to the number of malicious script injected for each category of context of attack vectors. Table 15 highlights the detection rate of both the platforms of multimedia web applications w.r.t. individual category of context of attack vectors.

It is clearly reflected from the table, the highest XSS worm detection rate is observed for BlogIt web application in the category of non-OSN-based web application. On the other hand, the highest XSS worm detection rate is observed in Humhub and Joomla. The next sub-section discusses the performance analysis and validation of the observed results of XSS-Secure on two different platforms of web applications.

4.3 Performance analysis

This sub-section discusses a detailed validation and performance analysis of our cloud-based framework by conducting two statistical analysis methods (i.e., F-Measure and F-test Hypothesis). We have also compared the XSS attack detection capability of our proposed framework with other recent XSS defensive methodologies based on some useful metrics. The analysis conducted reveal that our framework produces better results as compared to existing state-of-art techniques.

4.3.1 Performance analysis using F-Measure

For the binary classification, precision and recall are the values used for evaluations. And F-Measure is a harmonic mean of precision and recall.

Table 15 XSS detection rate (in %age) of non-OSN and OSN-based multimedia web applications

Non-OSN web applications	Wackopicko	Scarf	BlogIt	PHPBB v2	OsCommerce
Attack vector categories					
JSAV	86	89	94	86	91
UAV	80	80	90	80	85
HMT	87	93	80	80	80
CSSAV	91	82	86	95	86
HMEH	91	86	91	89	89
OSN Web applications	Humhub	Elgg	Drupal	Wordpress	Joomla
Attack vector categories					
JSAV	91	94	89	91	94
UAV	90	90	85	85	90
HMT	93	93	80	87	87
CSSAV	91	86	86	91	91
HMEH	97	91	94	91	94

$$False\ Positive\ Rate\ (FPR) = \frac{False\ Positves\ (FP)}{False\ Positives\ (FP) + True\ Negatives\ (TN)}$$

$$False\ Negative\ Rate\ (FNR) = \frac{False\ Negatives\ (FN)}{False\ Negatives\ (FN) + True\ Positives\ (TP)}$$

$$precision = \frac{True\ Positive(TP)}{True\ Positive(TP) + False\ Positive(FP)}$$

$$recall = \frac{True\ Positive(TP)}{True\ Positive(TP) + False\ Positive(FP)}$$

$$F{-}Measure = \frac{2(TP)}{2(TP) + FP + FN}$$

Here we calculate the precision, recall and finally F-Measure of observed experimental results of our framework on two different platforms of world web applications (i.e. non-OSN and OSN-based web applications). F-Measure generally analyzes the performance of system by calculating the harmonic mean of precision and recall. The analysis conducted reveals that our cloud-based framework exhibits high performance as the observed value of F-Measure in the two platforms of web applications in greater than 0.9. Table 16 highlights the detailed performance analysis of our proposed framework on five real world multimedia web applications and five OSN-based multimedia web applications. It is clearly reflected from the Table 16 that the performance of XSS-Secure on both the platforms of multimedia web applications is almost 97 % as the highest value of F-Measure is 0.970. In addition to this, the lowest False Negative Rate is observed in OsCommerce, Elgg, Drupal and Wordpress. This validates the performance of our framework.

4.3.2 Performance analysis using F-test

In order to prove that the number of malicious scripts detected (i.e. number of True Positives (TP)) is less than to the number of malicious scripts injected; we use the F-test hypothesis, which is defined as:

Table 16 Performance analysis of XSS-Secure by calculating F-measure

Non-OSN web application	Total	# of TP	# of FP	# of TN	# of FN	Precision	FPR	FNR	Recall	F-measure
Wackopicko	127	111	6	6	4	0.948	0.5	0.034	0.965	0.962
Scarf	127	110	6	6	5	0.948	0.5	0.043	0.956	0.958
BlogIt	127	114	5	6	2	0.931	0.533	0.035	0.964	0.954
PHPBB v2	127	110	6	7	4	0.948	0.6	0.035	0.964	0.962
OsCommerce	127	111	8	6	2	0.932	0.8	0.017	0.982	0.962
OSN-Based Web Application	Total	# of TP	# of FP	# of TN	# of FN	Precision	FPR	FNR	Recall	F-Measure
Humhub	127	116	5	4	2	0.958	0.5	0.016	0.983	0.970
Elgg	127	115	5	5	2	0.958	0.5	0.017	0.982	0.970
Drupal	127	112	7	6	2	0.941	0.538	0.017	0.982	0.961
Wordpress	127	114	6	5	2	0.950	0.545	0.017	0.982	0.967
Joomla	127	117	6	3	1	0.951	0.67	0.008	0.991	0.970

Null Hypothesis: H_0 = Number of malicious scripts detected is less than the number of malicious scripts injected. ($S1^2 = S2^2$)
Alternate Hypothesis: H_1 = Number of malicious scripts injected is greater than number of malicious scripts detected ($S1^2 < S2^2$)

The level of Significance is ($\alpha = 0.05$). The detailed analyses of statistics of XSS attack worms applied and detected are illustrated in the Tables 17 and 18. In our work, we utilized and injected total of 127 XSS attacks vectors from the freely available XSS attack repositories (as listed in the Table 6) in the ten web applications. But here note that, for evaluating and validating the performance of our framework by using F-test, we injected different number of malicious scripts in all the ten web applications.

# of Malicious Scripts Injected
# of Observation (N_1) = 10
Degree of Freedom dof (df_1) = N_1 -1 = 9.
# of Malicious Scripts Detected
# of Observation (N_2) = 9
Degree of Freedom dof (df_2) = N_2 -1 = 4.

$$F_{CALC} = S_1^2 / S_2^2 = 0.859$$

The tabulated value of F-Test at $df_1 = 9$, $df_2 = 9$ and $\alpha = 0.05$ is

$$F_{(df1,df2,1-\alpha)} = F_{(9,\ 9,\ 0.95)} = 3.1789$$

We know that the hypothesis that the two variances are equal (Null Hypothesis) is rejected if

$$F_{CALC} < F_{(df1,df2,1-\alpha)}$$

Table 17 Statistical analysis of malicious scripts injected

Web applications	# of malicious scripts injected (X_i)	$(X_1 - \mu)$	$(X_1 - \mu)^2$	Standard deviation $\mathbf{S_1} = \sqrt{\sum_{i=1}^{N1} (Xi-\mu)^2/(N1-1)}$
Wackopicko	120	1	1	2.472
Scarf	124	5	25	
BlogIt	122	3	9	
PHPBB v2	118	–1	1	
OsCommerce	119	0	0	
Humhub	121	2	4	
Elgg	117	–2	4	
Drupal	120	1	1	
Wordpress	116	–3	9	
Joomla	118	–1	1	
	Mean ($\mu = \sum Xi/N1 = 119$)		$\sum_{i=1}^{N1} (Xi-\mu)^2 = 55$	

Table 18 Statistical analysis of malicious scripts detected

Web applications	# of malicious scripts detected (X_i)	$(X_1 - \mu)$	$(X_1 - \mu)^2$	Standard deviation $\mathbf{S_2} = \sqrt{\sum_{i=1}^{N1} (Xi-\mu)^2/(N2-1)}$
Wackopicko	115	1	1	2.666
Scarf	118	4	16	
BlogIt	118	4	16	
PHPBB v2	112	–2	4	
OsCommerce	113	–1	1	
Humhub	114	0	0	
Elgg	112	–2	4	
Drupal	116	2	4	
Wordpress	111	–3	9	
	111	–3	9	
	Mean ($\mu = 2\sum Xi/N2 = 114$)		$\sum_{i=1}^{N1} (Xi-\mu)^2 = 64$	

Since $F_{CALC} < F_{(9,\ 9,\ 0.95)}$ therefore, we accept the alternate hypothesis (H1) that the first standard deviation (S_1) is less than the second standard deviation (S_2). Hence it is clear that the number of XSS worms detected is less than number of XSS attack vectors injected and we are 95 % confident that any difference in the sample standard deviation is due to random error.

4.3.3 Performance analysis using response time

For better analysis of performance analysis, we have also calculated the response time of our cloud-based framework in detection mode for the detection and sanitization of injected XSS worms in different frameworks and cloud platforms. Nowadays, several IT organizations install the setup of their web applications in the infrastructures of cloud for better response time efficiency. The same optimized response time was observed in both the modes of proposed framework deployed in the virtual machines of cloud environment. Table 19 highlights the response time of our proposed framework in different environment (i.e. without cloud infrastructure) and cloud platform.

XSS-Secure is a purely OSN cloud-based XSS defensive framework that detects and alleviates the XSS worms from the virtual machines deployed in the cloud computing environment. We tested the response time of XSS attack detection capability and sanitization performance on the cloud platforms as well as in other environments. The response time observed was low for all the two platforms of multimedia web applications deployed on the cloud settings in comparison to other settings of infrastructure.

4.4 Comparison

This section discusses the comparison of XSS-Secure with the other recent existing XSS defensive methodologies. Table 20 compares the existing client-side and sanitization-based

Table 19 Performance analysis using response time calculation

	Response time (in ms)			Response time (in ms)	
Non-OSN platforms	Without cloud platform	On cloud platform	OSN platforms	Without cloud platform	On cloud platform
Wackopicko	2096	2018	Humhub	2189	1983
Scarf	2312	2216	Elgg	3215	2498
BlogIt	2789	2676	Drupal	2318	2113
PHPBB v2	3126	2986	Wordpress	2419	2286
OsCommerce	2753	2697	Joomla	2517	2305

state-of-art techniques with our work based on eleven useful performance parameters: AMech: Analyzing Mechanism, TOA: Technique of Analysis, JSWDP: JavaScript Worm Detection Proficiency, FNR: False Negative Rate, WBM: Web Browser Modifications, CSSAN: Context-Sensitive Sanitization, CSRP: Context-Sensitive Runtime Parsing, JCMon: JavaScript Code Monitoring, JavaScript Code Modifications (JCMod), Automated Pre-Processing Required (APPR) and Support of Detection of Partial Injection of JS worms.

It is clearly reflected from the Table 20 that very few recent methodologies have focused on the two important categories of XSS worms (i.e., persistent and non-persistent). In addition, recognition of malicious script methods is simply evaded by most of the techniques. Moreover, lot of pre-processing is required in the existing framework of web applications for their successful execution on different platforms of web browsers as well as web applications. Context-aware sanitization is simply evaded by most of these existing sanitization-based techniques. Although, they perform the sanitization on the XSS attack vectors in a context-insensitive manner. Such sort of conventional sanitization methods are easily bypassed by the attackers. Most of the existing client-side techniques demand major modification in the source code of existing web browser.

Table 20 Comparison of existing work with XSS-secure

Techniques parameters	PathCutter [6]	Saner [2]	XSSFilt [24]	XSS auditor [3]	ScriptGard [27]	Livshits et al. [16]	XSS-secure (Our work)
AMech	Passive	Active	Passive	Active	Passive	Passive	Active
TOA	Dynamic	Static, Dynamic	Static	Static	Dynamic	Dynamic	Dynamic
JSWDP	Medium	Low	Low	Low	Medium	Low	High
FNR	High	Medium	High	Medium	Medium	Medium	Low
WBM	✓	✗	✓	✓	✗	✗	✗
CSSAN	✗	✗	✗	✗	✓	✗	✓
CSRP	✗	✗	✓	✓	✗	✗	✓
JCMon	✓	✓	✓	✓	✓	✓	✓
JCMod	✗	✓	✓	✗	✓	✓	✗
APPR	✓	✓	✗	✗	✓	✓	✗
SDPIJ	✗	✗	✓	✗	✗	✗	✓

On the other hand, we have also compared the performance analysis of existing client-side XSS filter (i.e. XSS-Auditor) with our cloud-based XSS-Secure. XSS-Auditor is installed as an extension of Google Chrome web browser. Here also, we have verified the XSS worm detection capability of XSS-Auditor by injecting 127 XSS worms on our five OSN-based web applications. Table 21 highlights the performance comparison of our cloud-based XSS-Secure with existing client-side XSS filter (XSS-Auditor). It can be clearly observed from the Table 21 that the value of F-measure is decreasing in all the platforms of OSN-based web applications for XSS-Auditor in comparison to our work.

Although, XSS-Auditor performs the sanitization on the untrusted variables of JavaScript in a context-aware manner, yet this filter is not capable enough to determine all possible contexts of such untrusted variables. Therefore, the sanitization of such variables in such malicious variables becomes ineffective for the XSS-Auditor. However, XSS-Secure is capable enough to determine the probable different context of malicious variables of JavaScript prior to the execution of context-sensitive sanitization procedure.

4.5 Reason behind generation of false negatives

We noticed some series of false negatives during the implementation and testing of our technique. However, all those discovered false negatives were just implementation-related flaws that were fixed later on. We have also utilized the capabilities of some advanced security researchers for discovering some extra false negatives. Luckily, we found a false negative interrelated to string decoding process. In reaction to this, we modified the similarity indicator component to disregard every non-ASCII characters. Such practices endorse that we possess small assertion that our technique lacks false negatives. We have continuously involved our security investigators for noticing extra rate of false negatives in the future. Numerous techniques have been continuously utilized by the researchers for detecting random code execution vulnerabilities in advanced code origins. Though, the proof clearly reflects that such false negatives would result from implementation faults, which could be fixed through updating at regular intervals.

Table 21 Performance comparison of XSS-Auditor with our work

Web application	# of TP	# of FP	# of TN	# of FN	Precision	Recall	F-measure
Cloud-based XSS-secure							
Humhub	116	5	4	2	0.958	0.983	0.970 ↑
Elgg	115	5	5	2	0.958	0.982	0.970 ↑
Drupal	112	7	6	2	0.941	0.982	0.961 ↑
Wordpress	114	6	5	2	0.950	0.982	0.967 ↑
Joomla	117	6	3	1	0.951	0.991	0.970 ↑
XSS-Auditor [3]							
Humhub	102	10	11	4	0.910	0.962	0.935 ↓
Elgg	109	6	6	6	0.947	0.947	0.947 ↓
Drupal	105	7	8	7	0.937	0.937	0.937 ↓
Wordpress	97	12	10	8	0.889	0.923	0.906 ↓
Joomla	101	9	8	10	0.918	0.909	0.914 ↓

4.6 Reason behind generation of false positives

To evaluate the rate of false positives, we installed our technique in the developer channel of Google Chrome and watched the clients of such web browser for filing error reports. We generally found some interesting errors at the initial iterations of execution of our technique. We were successfully able to fine-tune our technique for removing the false positive rate. The details of these false positives and method to remove them are described as follows:

- We found that the utilized open source web applications that comprises of < base > elements has encountered large rate of false positives. The < base > elements have been blocked by XSS-Secure because of the occurrence of base URL in the URL address of the web page. After encountering such problem, we have generated a whitelist of base URLs from the similar origin to remove the rate of false positive.
- We have also discovered that XSS-Secure disrupt the feature of chat on the utilized web applications as this feature includes an exterior script via URL address delivered in the form of query parameter. Although, the web server of these web applications confirms that the delivered URL address is pointing towards the web server handled by these web applications. We diminish the category of vulnerabilities, which we shield to eliminate insertions inside the src attribute for removing this false positive rate. We deployed this alteration by thwarting the loading of script content from an external script simply if each byte of src attribute is included in the request.

5 Conclusion and future work

Boom of multimedia-based social networking sites such as Facebook, Twitter, Linkedin, etc. has escalated numerous security and privacy issues of users on OSN. Worms like XSS are found to be one of the topmost threats in such networks. This article proposed XSS-Secure, a framework for the cloud platforms that preserves the privacy of online users against XSS worms on the platforms of OSN. The framework executes in two modes: Training and Detection Mode. The former mode generates the web templates corresponding to each extracted HREQ. Such web templates have explored for the malicious untrusted variables of JavaScript and inject the sanitization primitives in such location of variables. The sanitized untrusted JavaScript code is stored in the sanitizer snapshot repository for further reference in the online mode. Hereafter, the detection mode detects the injection of untrusted JavaScript variables by discovering their non-sanitized form in the generated HRES messages on the OSN server. Any injection of untrusted variable of JavaScript will be considered as a XSS worm. In addition, the online mode finds the context of such untrusted variables and performs the auto-context sensitive sanitization on such variables for their safe interpretation on the web browser of online user.

Ten tested open source platforms of multimedia-based web applications including the platforms of non-OSN and OSN-based web sites have utilized for evaluating the XSS attack detection and alleviation capability of XSS-Secure. The evaluation outcomes highlight that our

framework XSS detection rate suffer from low and acceptable rate of false negative, false positive and less performance overhead. We will also try to evaluate the capabilities of our framework on more real world platforms of OSN and will detect the exploitation of other categories of XSS attack as a part of our further work.

Acknowledgments The authors would like to thank members of Information and Cyber Security Research Group working in the National Institute of Technology Kurukshetra, India for their valuable feedback and worthwhile discussions. This work was financially supported by TEQIP-II.

References

1. Almorsy M, Grundy J, Mueller I (2010) An analysis of the cloud computing security problem. Proc 2010 Asia Pacific Cloud Workshop, Colocated with APSEC2010, Australia
2. Balzarotti D, Cova M, Felmetsger V, Jovanovic N, Kirda E, Kruegel C, Vigna G (2008) Saner: composing static and dynamic analysis to validate sanitization in web applications. In Sec Privacy, 2008. SP 2008. IEEE Symp:387–401. IEEE
3. Bates D, Barth A, Jackson C (2010) Regular expressions considered harmful in client-side XSS filters. Proc World Wide Web: 91–100
4. Blogit. Available at: http://www.blogit.com/Blogs/
5. Byong JH, Jung I-Y, Kim K-H, Lee D-k, Rho S, Jeong CS (2013) Cloud-based active content collaboration platform using multimedia processing. EURASIP J Wireless Commun Networking (JWCN), Springer, 2013:63
6. Cao Y, Yegneswaran V, Porras PA, Chen Y (2012) PathCutter: severing the self-propagation path of XSS javascript worms in social web networks. NDSS
7. CVE Details (2013) Vulnerabilities by type. Retrieved from http://www.cvedetails.com/vulnerabilitie-by-types.php
8. Drupal social networking site. Available: https://www.drupal.org/download
9. Elgg social networking engine. Available at: https://elgg.org
10. Gupta S, Gupta BB (2014) 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, 174–191, USA
11. Gupta S, Gupta BB (2015) Cross-site scripting (XSS) attacks and defense mechanisms: classification and state-of- art. Int J Syst Assurance Eng Manag, Springer
12. Gupta S, Gupta BB (2015) XSS-SAFE: a server-side approach to detect and mitigate cross-site scripting (XSS) attacks in JavaScript code. Arab J Sci Eng: 1–24
13. Gupta S, Gupta BB (2016) Automated discovery of javascript code injection attacks in PHP web applications. Proc Comput Sci 78:82–87
14. Gupta BB, Shashank G, Gangwar S, Kumar M, Meena PK et al (2015) Cross-site scripting (XSS) abuse and defense: exploitation on several testing bed environments and its defense, special issue of secured communication in wireless and wired networks. J Inform Privacy Sec, Taylor & Francis Online 11(2):118–136
15. Gupta MK et al. (2015) XSSDM: towards detection and mitigation of cross-site scripting vulnerabilities in web applications. Adv Comput, Commun Inform (ICACCI), 2015 Int Conf. IEEE
16. Hooimeijer P, Livshits B, Molnar D, Saxena P, Veanes M (2011) Fast and precise sanitizer analysis with BEK. Proc 20th USENIX Conf Security: 1-1. USENIX Association
17. Humhub social networking site. Available at: https://www.humhub.org/en
18. Jabbar S, Naseer K, Moneeb G, Rho S, Chang HB (2016) Trust model at service layer of cloud computing for educational institutes. J Supercomput (JoS), Springer 72(1):247–274
19. Joomla social networking site. Available at: https://www.joomla.org/download.html
20. Myspace samy worm [online]. Available: http://namb.la/popular/tech.html
21. Orkut and Twitter XSS worm [online]. Available: http://www.xssed.com/news/120/Twitter_and_Orkut_XSS_worms_in_the_news/
22. OsCommerce. Available at: http://www.oscommerce.com/

23. Parameshwaran I et al. (2015) DexterJS: robust testing platform for DOM-based XSS vulnerabilities. Proc 2015 10th Joint Meet Found Software Eng. ACM
24. Pelizzi, Riccardo, and R. Sekar. "Protection, usability and improvements in reflected XSS filters." In ASIACCS, p. 5. 2012.
25. phpBB v2. Available at: http://sourceforge.net/projects/phpbb/files/phpBB%202/phpBB%20v2.0.23/
26. Rsnake (2008) XSS Cheat Sheet. http://ha.ckers.org/xss.html
27. Saxena P, Molnar D, Livshits B (2011) SCRIPTGARD: automatic context-sensitive sanitization for large-scale legacy web applications. Proc 18th ACM Conf Comput Commun Sec: 601–614. ACM
28. Stock B et al. (2015) From facepalm to brain bender: exploring client-side cross-site scripting. Proc 22nd ACM SIGSAC Conf Comput Commun Sec. ACM
29. Wackopicko. Available at: https://github.com/adamdoupe/wackopicko
30. Weinberger J, Saxena P, Akhawe D, Finifter M, Shin R, Song D (2011) A systematic analysis of XSS sanitization in web application frameworks. Comput Sec–ESORICS 2011:150–171. Springer Berlin Heidelberg
31. Wordpress. Available at: https://wordpress.org/
32. Xiao W et al. (2014) Preventing client side XSS with rewrite based dynamic information flow. Parallel Architect, Algorit Prog (PAAP), 2014 Sixth Int Symp. IEEE

SHASHANK GUPTA He was born on 25th September, 1987 in the state of Jammu and Kashmir, India. Presently he is pursuing PhD in the Department of Computer Engineering from National Institute of Technology Kurukshetra, Haryana, India. He has completed M.Tech.in the Department of Computer Science and Engineering Specialization in Information Security from Central University of Rajasthan, Ajmer, India. He has also done his graduation in Bachelor of Engineering (B.E.) in Department of Information Technology from Padmashree Dr. D.Y. Patil Institute of Engineering and Technology Affiliated to Pune University, India. He has numerous publications in peer reviewed International Journals including several SCIE-Indexed journals, numerous publications in International Conference Proceedings along with several book chapters. He is also a student member of IEEE and ACM. His area of interest includes Web Security, Online Social Network Security, Cloud Security and theory of Computation.

Dr. B.B. Gupta received PhD degree from Indian Institute of Technology Roorkee, India in the area of Information and Cyber Security. In 2009, he was selected for Canadian Commonwealth Scholarship and awarded by Government of Canada Award ($10,000). He spent more than six months in University of Saskatchewan (UofS), Canada to complete a portion of his research work. He has published more than 70 research papers (including 01 book and 08 chapters) in International Journals and Conferences of high repute including IEEE, Elsevier, ACM, Springer, Wiley Inderscience, etc. He has visited several countries, i.e. Canada, Japan, China, Malaysia, Hong-Kong, etc. to present his research work. His biography was selected and publishes in the 30th Edition of Marquis Who's Who in the World, 2012.

He is also working principal investigator of various R&D projects. He is also serving as reviewer for Journals of IEEE, Springer, Wiley, Taylor & Francis, etc. He is serving as guest editor of various Journals. He was also visiting researcher with Yamaguchi University, Japan in 2015 and with Guangzhou University, China in 2016, respectively. At present, Dr. Gupta is working as Assistant Professor in the Department of Computer Engineering, National Institute of Technology Kurukshetra, India. His research interest includes Information security, Cyber Security, Mobile/Smartphone, Cloud Computing, Web security, Intrusion detection, Computer networks and Phishing.