Computers & Security 118 (2022) 102710

Contents lists available at ScienceDirect

Computers & Security

journal homepage: www.elsevier.com/locate/cose

Securing heterogeneous embedded devices against XSS attack in intelligent IoT system

Pooja Chaudhary[a], Brij B. Gupta[b,c,d,*], A.K. Singh[a]

[a] Department of Computer Engineering, National Institute of Technology Kurukshetra, India
[b] Department of Computer Science and Information Engineering, Asia University, Taichung 413, Taiwan
[c] Research and Innovation Department, Skyline University College, Sharjah P.O. Box 1797, United Arab Emirates
[d] Macquarie University, Australia

ARTICLE INFO

Article history:
Received 6 September 2021
Revised 22 March 2022
Accepted 28 March 2022
Available online 30 March 2022

Keyword:
Intelligent IoT System
Embedded device security
Firmware security
Device management web interface vulnerabilities
Cross-site scripting (XSS) attack
Web application vulnerability

ABSTRACT

Today, we are living in the realm of Internet of Things (IoT) where simple objects are embedded with the capabilities to understand and operate in its surroundings for offering distinct services to the users. These objects are shipped with their user interfaces that facilitate user to perform administrative activities on the devices using a web browser linked to the device's server. Cross-Site Scripting (XSS) is the most prevalent web application's vulnerability, exploited by an attacker to compromise the embedded devices. This research work is focused towards the development of an approach to defend against XSS attack to safeguard embedded devices deployed in intelligent IoT system. It performs identification through comparing injected strings with the blacklisted attack vectors and mitigates its harmful effects by implementing filtering method in an optimized fashion. It is a fog-enabled approach that operates locally to identify the compromised device within the IoT network. We demonstrate attack exploitation on two smart devices including digital IP Camera and wireless router and then tested the performance of our proposed approach on them. The experimental results highlight the efficacy of the approach as it attains an accuracy of 0.9 and above, on both the tested platforms.

1. Introduction

Internet of Things (IoT) exemplifies the vision of ubiquitous computing where there is an intelligent communication between humans and embedded systems to accomplish complex tasks in multitude of intelligent environments comprising smart home, smart office, smart cities and smart healthcare (Tewari, A., & Gupta, B. B, 2020"). IoT networks typically represents a network of resource-constrained "things" or smart objects that operate collaboratively to achieve pre-defined objectives by the application. Intelligent IoT system symbolizes internet-enabled embedded systems facilitating real-time computing with low-energy depletion and low-maintenance. These systems are intricate because heterogeneous devices (or embedded devices) coordinate and work together within the network to acheve the designated task (Hameed, Khan, and Hameed, 2019). Recently, embedded devices including smart cameras, wireless routers, and wireless printers are offered with their management interface that enables device configuration by communicating with device server with the help of web browser. It is delivered to allow user to directly interact with the device employing a user interface, instead of learning a command-line language for the device. It also helps the vendor as there is no need to issue a client-side software.

Although, it seems to be an efficient way, however, there are security threats involved due to the presence of latent vulnerabilities in the web applications (Hameed, Khan, and Hameed, 2019, Salhi, Tari, and Kechadi, 2021). One such commonly found and threatened web application vulnerability is Cross-Site Scripting (XSS) (Chaudhary and Gupta, 2018). XSS attack is triggered by injecting attack strings into the web application, so that whenever a user requests the infected web page then it gets executed by the browser. It is necessary to detect XSS in embedded devices because XSS exploit can lead to further massive scale attacks comprising (Chaudhary, Gupta, and Gupta, 2016):

- Unauthorized access to sensitive data.
- Redirecting the user to attacker control web site.
- Device exploitation in building botnet army for Distributed Denial of Service (DDoS) (Cvitić, Peraković, Gupta, and Choo, 2021; Dahiya and Gupta, 2021; Gupta et al., 2021).
- Reconfiguration of device's settings.

* Corresponding author.
E-mail address: gupta.brij@gmail.com (B.B. Gupta).

https://doi.org/10.1016/j.cose.2022.102710
0167-4048/© 2022 Elsevier Ltd. All rights reserved.

It is clear that XSS has annihilating consequences, thereby, it is imperative to develop solutions to secure embedded devices from the harmful effects of the XSS attacks. In this paper, we design an approach to detect and mitigate XSS attack in the intelligent IoT systems to shield embedded devices. Moreover, to reduce latency and bandwidth consumption, we utilize the benefits of fog computing environment (Dastjerdi and Buyya, 2016). It enables detection of vulnerable device in the local network at faster rate than in the centralized system. This approach defends against store and reflected XSS attack by comparing the injected attack strings with the blacklisted XSS attack vectors. Shielding against XSS attack guarantees a) protection of user's device credentials to hamper device control, b) no unauthorized malicious software or firmware download, c) no leakage of private information. The key contributions of this paper are outlined as follows:

- To design an approach to defend against stored and reflected XSS attack in fog-based intelligent IoT system infrastructure.
- To extract and compare query string parameter values with blacklisted attack vectors using Boyer-Moore string matching algorithm, to detect reflected XSS.
- To construct parse tree, mine maliciously injected attack strings and compare with debarred attack strings using Boyer-Moore string matching algorithm, to identify stored XSS.
- To exploit known XSS vulnerability in Hitron CODA 4582u router and Bosch Flexidome IP indoor 5000 HD camera, for the purpose of attack demonstration.
- To assess the attack detection efficiency of proposed approach on the exploited devices and examine the performance using prominent metrices comprising precision, recall, F-measure, and FPR.

This paper schema is as follows: Section 2 illuminates a simple scenario of XSS attack in intelligent IoT system. Section 3 highlights recent state-of-art techniques developed by other researchers in the field. Section 4 elucidates the proposed approach exhaustively, followed by implementation and experimental results discussion in Section 5. Lastly, Section 6 summarizes the paper.

2. XSS attack in IoT system

XSS attack hampers the security of the web applications that enables an attacker to inject attack payload in the input field present in the vulnerable web application (Grossman et al., 2007). The major cause of this attack is the improper filtering of the data provided by any user which is exploited by an attacker for illicit code injection. Most often, device management web interface of the embedded devices is vulnerable to two main types of XSS attack: persistent and reflected XSS attack (Rodríguez, Torres, Flores, and Benavides, 2020). In persistent XSS, an attacker inserts malicious attack string into the web site that permanently persists in the database of the web site. Thus, whenever a user requests the infected web page then server will generate the response with embedded attack string and eventually gets rendered by the web browser. In reflected XSS, attacker crafts URL with malicious string and sent it to user so that when the victim clicks on it and makes a request to application server then script reflects back to browser, resulting in script execution. This fulfills the evil intentions of the attacker such as cookie stealing, privilege escalation, malicious firmware download, and so on. A simple scenario of XSS attack is depicted in Fig. 1.

3. Related work

This section shed some light on the major contributions by different researchers in the field of XSS detection and mitigation. Various mechanisms have been adopted to defend against XSS attack on different platforms comprising social networking platform, CMS web application, mobile apps, and so on. However, little consideration has been paid to the XSS alleviation from embedded devices deployed in IoT networks. Thus, we discuss some of the useful methods for XSS detection in diverse areas.

Solutions have been proposed based on the static and dynamic analysis of the infected web page. In (Jovanovic, Kruegel, and Kirda, 2006, May), authors have designed an open-source tool, named as Pixy for detecting XSS vulnerability in PHP-based web applications through context-sensitive data flow analysis. Salas et al. (Salas and Martins, 2014) have designed an approach on the basis of two methods comprising penetration testing and fault injection to detect XSS attack against web services. Gupta et al. (Gupta and Gupta, 2016) designed a framework, XSS-SAFE, which is a server-side framework to prevent against XSS attack by performing the comparison between genuine web page features and observed features of the generated response web page. Duchene et al. (Duchene, Rawat, Richier, and Groz, 2014, March) designed a black-box XSS fuzzer named as KameleonFuzz, that has the ability to generate malicious input to test for existing XSS vulnerability in the web applications, using genetic algorithm with attack grammar. It is basically a taint-flow analysis-based technique for attack detection. Ahmed et al. (Ahmed and Ali, 2016) developed an approach based on the idea of taint-flow checking. It uses genetic algorithm along with XSS attack patterns for the generation of test data which is used to identify all possible script flow in the web page. It is achieved to identify the successful execution of the XSS attack by examining each possible path covered by the scripts. Gupta et al. (Gupta, Chaudhary, and Gupta, 2020) designed a XSS defensive framework for protecting users in smart city. They have used two approaches for achieving this task: first, they perform vulnerable taint tracking for protecting dynamic web pages and second, applied trusted marks for shielding static web pages. Wang et al. (WANG, GU, and ZHAO, 2017) designed an approach for XSS detection using hidden Markov model that is used to generate dynamic attack vectors and is evaluated with its similarities in requests. Moreover, due to the advancements in AI technology, various machine leaning-based techniques have been proposed to detect XSS attack in web applications (Rathore, Sharma, and Park, 2017, Fang, Li, Liu, and Huang, 2018, March, Mokbal et al., 2021, Zhou and Wang, 2019, Mokbal et al., 2019, Zhang et al., 2020, Mani, Moh, and Moh, 2021). Because XSS attack is used as a stepping stone to trigger other cyber-attacks such as Distributed Denial of Service (DDoS) which has gained much attention by the research community (Boyer and Moore, 1977, Html 2022).

3.1. Research issues in existing literature

Multitude of defensive techniques have been designed to alleviate XSS attack. Nevertheless, there are some security challenges that needs to be incorporated in the futuristic approaches to handle XSS vulnerability:

- Static and dynamic analysis-based methods are not efficient to handle new XSS attack vector due to increased complexity and incurs high processing overhead. Most of the techniques demand source code modifications to mitigate XSS attack, resulting into increased processing time and performance surplus. Furthermore, very less techniques have been designed to shield embedded devices from the XSS attack.
- Most of the existing techniques are not capable to identify attack strings crafted using new features of web development programming languages like HTML5. Attackers use new tags and attributes such as iframe, video, autofocus, to create attack vectors to bypass XSS filters in the modern web browsers. Thus, it is imperative to design such solutions that possess the ca-

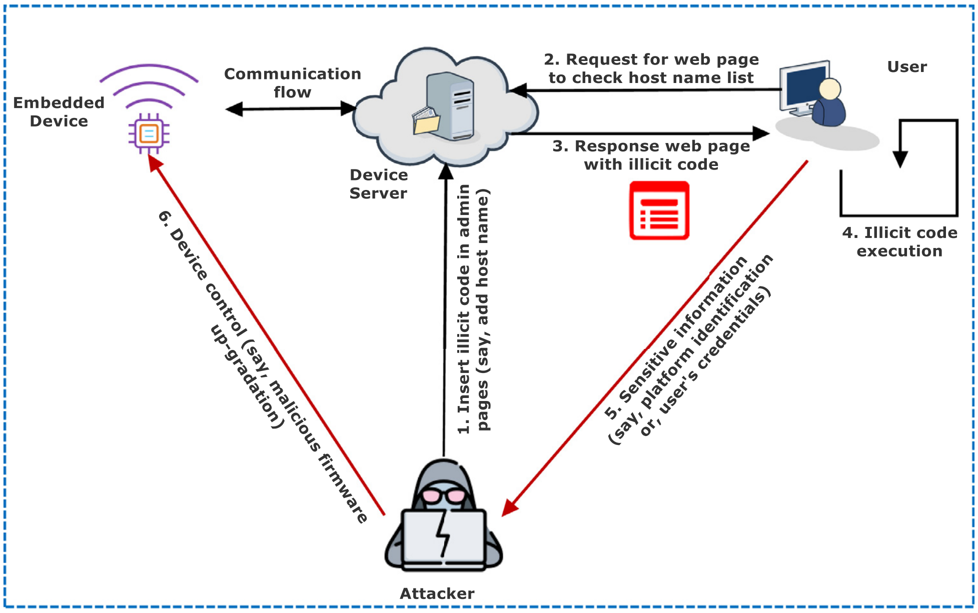

Embedded Device
Communication flow
Device Server
2. Request for web page to check host name list
User
3. Response web page with illicit code
4. Illicit code execution
1. Insert illicit code in admin pages (say, add host name)
6. Device control (say, malicious firmware up-gradation)
5. Sensitive information (say, platform identification or, user's credentials)
Attacker

Fig. 1. XSS attack scenario in IoT system.

pability to recognize such attack vectors to secure against XSS attack.

- Sanitization of the attack strings is considered to be the foremost method to mitigate its effects. However, sanitization without context determination or improper context determination is not an efficient way to shield against XSS attack as placement and selection of correct sanitization routine depends on the proper identification of the context of the attack vectors. Additionally, nested context determination is also obligatory for efficient sanitization of the attack vectors. Thus, it is crucial for the defensive solutions to focus on precise determination of the context and implementation of correct sanitization routines.
- More often, attacker injects similar attack strings at multiple locations in the web application and most of the defensive methods filtered out each injected attack vector separately. It is a time intensive filtering as it performs same kind of operations repeatedly. Thus, it is mandatory to develop solutions that accomplish filtering operation in an optimized manner.
- Existing state-of-art approaches identify XSS attack vectors by directly comparing them with unambiguous attack strings. However, attacker circumvent these techniques through crafting complex attack vectors. Attacker may use obfuscation of the attack strings and/or may perform partial script injection to achieve his evil intentions. Therefore, it is vital to incorporate methods in the defensive approaches to unveil these complex attack vectors.
- XSS detection competence of existing literature is usually tested and evaluated on the single source XSS dataset. This may hinder the detection efficacy of the approach against the new and complex attack vectors. Therefore, robust solutions should focus on the dataset extracted from multiple sources and it is also recommended to automate this procedure to identify more complex and new attack vectors quickly.

Motivated by such challenges, this paper presents an approach to secure embedded devices against XSS attack in the IoT networks. It operates by comparing the injected attack strings with the XSS attack vectors using string matching algorithm. If string is matched then it provides a potential alarm of XSS, otherwise it is considered as safe response. If XSS is indicated then, we identify the context of the injected script. To optimize the performance, we perform grouping of the injected attack strings on the basis on their context and content. Finally, filtering is applied to neutralize the script effects.

4. Proposed approach

In this section, authors exhaustively describe the proposed XSS defensive approach to protect the embedded devices deployed in intelligent IoT system. It shields against the stored and reflected XSS attack that is caused because of latent injection vulnerability in the device management web interface.

4.1. Conceptual design outline

The primary reason of instigating the XSS attack is the unvalidated execution of malicious attack strings in the user's browser, resulting into unauthorized actions. Hence, the prime goal of this research work is to obstruct the execution of attack strings in the user's browser by neutralizing its effects. Fig. 2 represents the conceptualized design view of the proposed approach. To overcome the limitations of centralized system-based solutions, we implement our approach at the fog nodes. It is achieved to perform attack detection locally in the IoT networks that ease the identification of vulnerable device and its web application. There are three key objectives of the proposed approach to identify and mitigate the effects of XSS attack strings. These are defined below:

- ***Objective 1:*** To identify reflected XSS attack, our approach operates by determining the attack strings present in the parameter value of the HTTP request URL. This problem can be formulated as:

  Consider $Q = \{q_1, q_2 \ldots q_n)$ *is the set of all parameters values present in the query string of the URL and* $AS = \{as_1, as_2 \ldots as_m\}$ *denotes the class of attack strings then identify all* y_i *such that* $y_i \in Q$ *and* $y_i \subseteq AS$.

- ***Objective 2:*** To identify stored XSS attack, our approach operates by parsing the input values present at each vulnerable location in the response web page and comparing them with the attack strings. This problem can be formulated as:

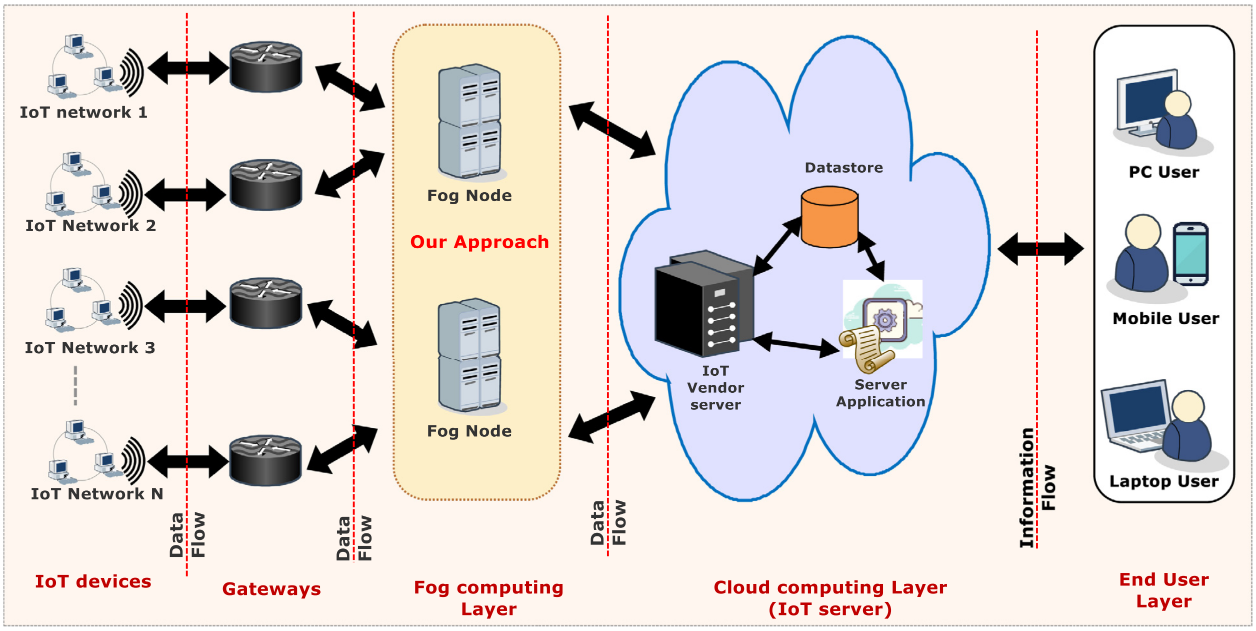


Fig. 2. Abstract Design Overview of the proposed approach.

Consider I = {i_1, i_2... i_n}is the list containing the injected input values at some vulnerable location L = {l_1, l_2... l_m}in the HTTP response web page and AS = {as_1, as_2... as_m} denotes the class of attack strings then identify all w_i and x_i such that ($w_i \in I$ and $x_i \in L$: w_i present at x_i) and ($w_i \subseteq AS$)

- ***Objective 3:*** To nullify the effects of XSS attack strings, our approach performs filtering of the identified attack strings, in the HTTP response web page, in an optimized manner. It performs grouping of the similar strings on the basis of their identified context like JS-based strings, HTML- based strings and their content also.

4.2. Detailed design overview

This section exhaustively demystifies the components involved in the functioning of our proposed approach. Fig. 3 illustrates the detailed design view of the proposed approach. We have defined each component, in detail, in the following sub-sections. Fig. 4 picturized the working flow of our approach to apprehend it more clearly and precisely.

4.2.1. Query string parameter extractor

This component assists in the identification of reflected XSS attack. Since the malicious script is embedded in the requested URL that reflects back in the response URL, thus, this component seizes each user's command i.e., HTTP request and correspondingly generated server's response i.e., HTTP response. Then, it extracts the parameter values included in the query string of the URL that may contain malicious payload. Mostly attacker apply some form of encoding methods such as HTML encoding, URL encoding, etc. to evade XSS filters at the client side. For instance, a simple malicious script *<script>alert("XSS attack")</script>* may be encoded as *%3Cscript%3Ealert%28%22XSS%20attack%22%29%3C%2Fscript%3E*. So, web browsers cannot identify the hidden meaning of this encoded value which gets eventually executed in the browser.

Therefore, it performs decoding of the extracted scripts to convert them into their original format so that it can be monitored easily. All the decoded extracted scripts are then forwarded to the string analyser for attack detection.

Table 1
List of malicious contexts in HTML page.

Elements	Context
HTML	PCDATA
	RCDATA
	CDATA
	Tag name
	Attribute name
	Attribute value: Quoted
	Attribute value: Unquoted
	Event attribute
	Tag text: String
	Attribute value: String
JavaScript	Method name
	Method value: REGEX
URL	Query: String

4.2.2. Parser

This component receives the generated HTTP response web page and constructs parse tree of the received web page. We employ html5lib parser to achieve this functionality because it provides broad optimal representation of the web page as it renders the web page in the similar manner as a web browser does. The generated parsed tree is utilized for extracting injected attack strings. To make it more understandable, we have created a dummy example as shown in listing 1. Here, there are two vulnerable locations $_GET('name') and $_GET('age') where an adversary may inject malicious scripts to instigate XSS attack. Fig. 5 depicts the constructed parsed tree for the code shown above.

4.2.3. Vulnerable variable detector & text value extractor

It receives the generated parse tree of the response web page and identifies the vulnerable locations (or variables) where an attacker may insert malicious attack strings. This is achieved by analysing the input values at each known vulnerable context of the HTML page, as shown in Table 1.

Furthermore, it extracts the text value present at each identified vulnerable location and store them into a list. Algorithm 1 represents the procedure implemented for achieving the objective of this component.

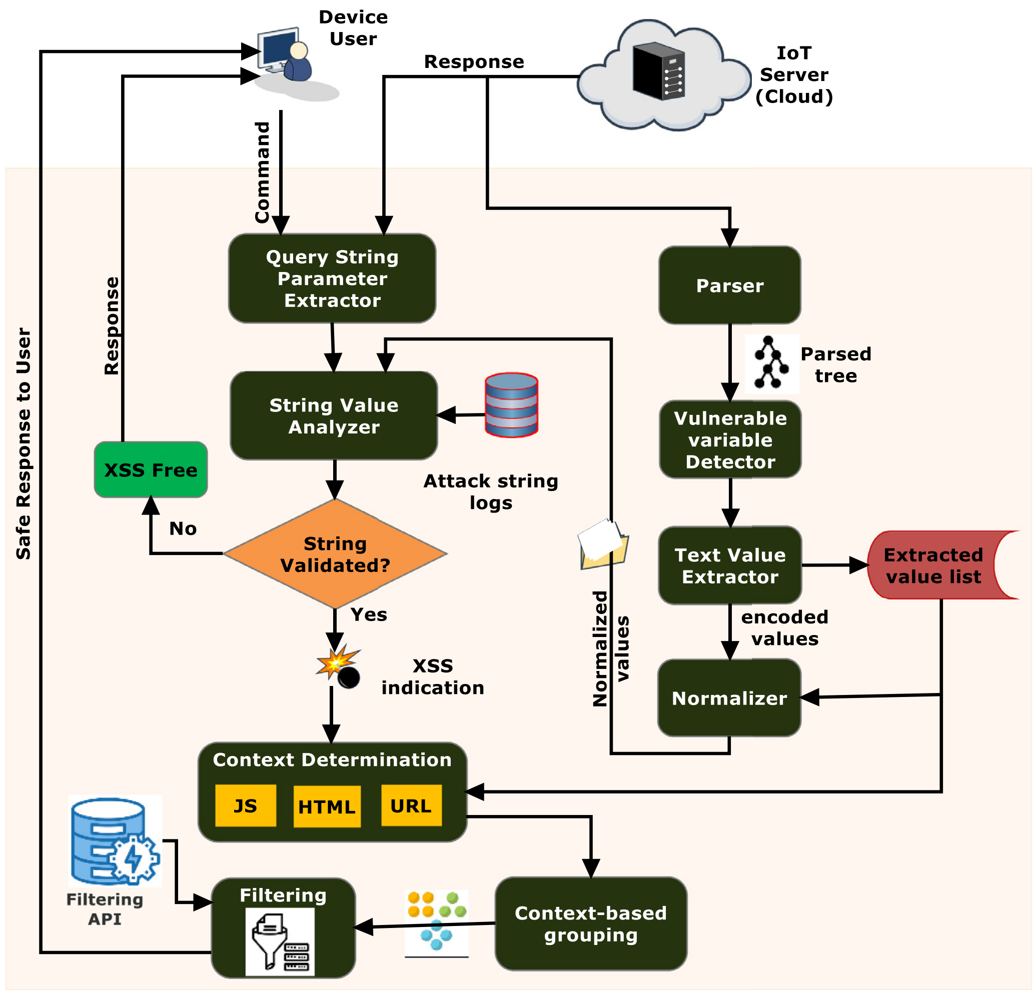


Fig. 3. Detailed Design view of the proposed approach.

This algorithm works as follows: it receives parse tree P(N, E) as it input. For each node (n_i) in the parse tree, we monitor 'n_i' value and compares it with the known vulnerable list of HTML Tags (H_Tag), HTML attributes(H_attr), HTML event (H_event) and URL properties (URL_prop). Since, JavaScript code is embedded with the help of HTML tag, thus, we analyse the tag value for JS features (JS_prop) and methods (JS_fun) that are used for drafting XSS attack strings. Finally, it combines all the extracted value into a single attack vector list (X_{PV}). This list is then forwarded to the next component i.e., normalizer for further processing.

4.2.4. Normalizer

This component receives the extracted vulnerable strings and normalizes them. Specifically, it performs decoding of these values. Attackers mostly applied some form of encoding techniques to obfuscate malicious scripts aiming to circumvent XSS filters present at the client-side. In order to restrict their execution in browser, it is inevitable to normalize or decode injected malicious scripts. Table 2 highlights types of encoding an attacker can apply for most commonly used special character "<".

We have also highlighted some other encoding codes for most frequently used symbols in attack strings, as shown in Table 3. For instance, a simple malicious script *<script>alert("XSS attack")</script>* may be encoded as *%3Cscript%3Ealert%28%22XSS%20attack%22%29%3C%2Fscript%3E.* After implementing this component, this script converts back into its original format, reflecting its hidden intent.

4.2.5. String value analyzer

This component performs the key task of our proposed approach i.e., attack detection. It receives the injected XSS attack strings from two sources: first, the extracted scripts from the parameters values of query strings in the HTTP request and HTTP response that is produced as an output of query string parameter extractor component. Second, it receives normalized attack strings that are captured from each vulnerable location in the response web page. It is generated as an output of normalizer component.

This component leverages blacklisted attack strings logs for attack detection. It parallelly compares the received vulnerable string value with the blacklisted scripts with the Boyer-Moore algorithm (Boyer and Moore, 1977) which is a fast string searching algorithm and efficiently works for the patterns which are lengthier as it performs some pre-processing steps that helps in avoiding character-wise searching. If strings found a match in blacklisted logs, then it indicates a potential signal of XSS attack, otherwise, it is a safe response and sent it to the user.

4.2.6. Context determination

The key objective of this component is to recognize the context of each injected malicious attack string. It is achieved to select and execute correct attack filter APIs. To complete this task, initially, we identify the context of vulnerable variable that is used for script injection. Then, the identified context will be applicable to the injected script also. For example, if attacker injects *<IMG SRC='javascript:alert("RSnake says, 'XSS'")'>* then he/she has

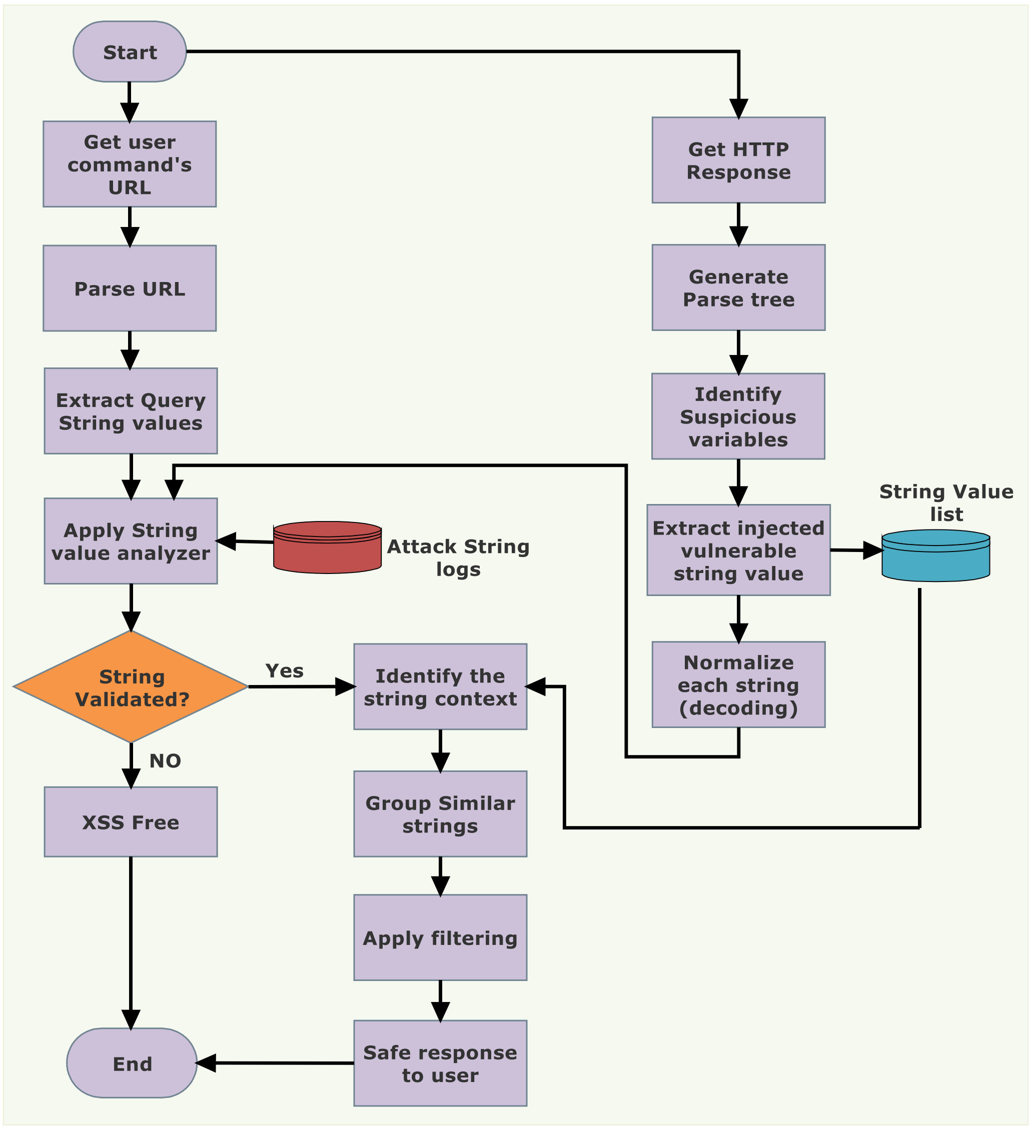


Fig. 4. Flow chart of the proposed approach.

exploited HTML <img> tag. Thus, the identified context is HTML tag and attribute value: quoted. Some of the identified context in the HTML web page is also shown in Table 1. Algorithm 2 is implemented for unveiling the context information.

4.2.7. Context-based grouping

Most often, attacker injects similar kinds of malicious attack strings at different vulnerable locations in the web page. Therefore, this component performs grouping of such kind of scripts on the basis of both the identified context and script content. Since we store extracted scripts in different lists as per their context information. For example, we store all HTML based attack string in "*H*" list (from Algorithm 1), JavaScript based strings in *"JS"* list, etc. To achieve content-based grouping, we implement Levenshtein-distance to measure the difference between attack string for each list discretely. Suppose two similar scripts as shown below:

```
<script>alert(48a$bc);</script>
<script>alert(48xv&ez);</script>
```

These scripts share similar pattern except their argument values. Therefore, a grouped template is generated through replacing the unsimilar characters with some token values, say, 'S' for alphanumeric characters and 'N' for numeric characters. Thus, template for above mentioned scripts is represented as: `<script>alert(48-S-);</script.` It enhances the attack mitigation efficiency of the proposed approach.

4.2.8. Filtering

This component restricts the execution of injected scripts in the web page. It receives the grouped attack strings along with their context information. It basically sanitizes the attack strings with the help of filtering APIs. It produces safe response and delivers it to the user. Algorithm 3 highlights the steps executed to fulfil the functionalities of context-based grouping and filtering. As we have grouped attack strings as per their identified context, thus, filtering is also applied in the similar manner. Here, we store the grouped template for each identified context (*H, JS, and U*) into '*m*' using levenshtein-distance. β is the threshold which determines the quality of similarity measures between two attack strings. If observed levenshtein-distance is greater that β then we accept the pair of strings as similar scripts. Otherwise, we will reject the pair and select another pair. Additionally, *F_API* represents the available APIs for filtering that are applied to the grouped templet stored in

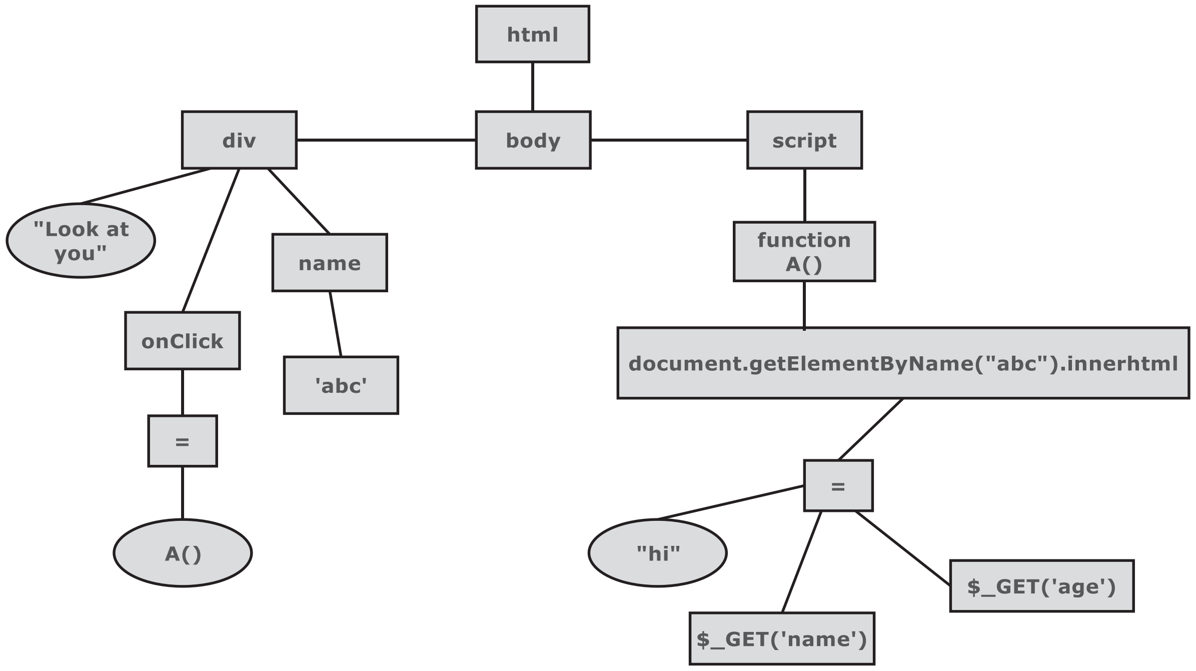


Fig. 5. Parse tree for code snippet shown in listing 1.

Algorithm 1
Vulnerable variable detection and value extraction.

```
Input: parsed tree of response web page P(N, E)
Output: Vulnerable string payload vector ($X_{PV}$)
H_tag ← HTML vulnerable tags list
H_att ← HTML vulnerable attribute list
H_event ← HTML vulnerable event handler list
JS_fun ← JS vulnerable function list
JS_prop ← JS vulnerable properties list
URL_prop ← URL vulnerable properties list
Start
H{}← ∅;
JS{}← ∅;
U{}← ∅;
$X_{PV}${}← ∅;
//Extract every tag, attribute and event handler from parse tree
For each node $n_i \in$ P(N, E) do
If ($n_i$.matches(H_att)) then
H ← H ∪ $n_i$.value;
elseif (ni.matches(H_event)) then
H ← H ∪ $n_i$.value;
elseif (ni.matches(H_tag)) then
H ← H ∪ $n_i$.value;
//collect JS string values from every possible place
If (($n_i$.value ∈ JS_prop) && ($n_i$.value ∈ JS_fun)) then
JS ← JS ∪ $n_i$.value;
End if
elseif (ni.matches(URL_prop) then
U ← U ∪ $n_i$.value;
End if
End for
$X_{PV}$ ← H ∪ JS ∪ U;
Return vulnerable string payload vector $X_{PV}$;
End
```

Algorithm 2
Identification of context.

```
Input: Extracted vulnerable payload vector (H, JS, U)
Output: Context information of each payload vector
Context token: $CT_1|CT_2|...|CT_N$;
H ← extracted vulnerable HTML values;
JS ← extracted vulnerable JS values;
U ← extracted vulnerable URL values;
Start
Temp ← Ø
C ← Ø;
//insert context token for each vulnerable variable in H, JS, U
For each $h_i$ ∈ H do
C ← CT($h_i$);
Temp ← Temp ∪ C;
End for
For each $Js_i$ ∈ JS do
C ← CT($Js_i$);
Temp ← Temp ∪ C;
End for
For each $u_i$ ∈ U do
C ← CT($u_i$);
Temp ← Temp ∪ C;
End for
//determine the type of inserted context token
For each $C_i$ ∈ Temp do
$X_I$ ← extract value X from CT(X) for $C_i$;
//X is the placeholder for variable.
If ($X_I$ ∈ STRING) then
$\Gamma \mapsto CT_I$: String;
Else if ($X_I$ ∈ NUMERIC) then
$\Gamma \mapsto CT_I$: Number;
Else if ($X_I$ ∈ REGEX) then
$\Gamma \mapsto CT_I$: Regular expression;
Else if ($X_I$ ∈Quoted Data) then
$\Gamma \mapsto CT_I$: Quoted Data;
Else if ($X_I$ ∈ PCDATA) then
$\Gamma \mapsto CT_I$: Parsed character data;
Else if ($X_I$ ∈ CDATA) then
$\Gamma \mapsto CT_I$: Character data;
End If
Temp ← modified(CT);
End for
Return Temp;
End
```

'*m*' as per their context '*C*'. Then, the modified safe response web page is forwarded to the user.

Hence, our proposed approach efficiently shields against store and reflected XSS attack that are commonly identified in the device management web interface of embedded devices. Algorithm 4 and Algorithm 5 illustrates the mechanisms to alleviate stored and reflected XSS attack respectively.

Algorithm 3
Grouping and Filtering.

```
Input: context information and extracted string value
Output: modified response ($H_M$')
H ← extracted vulnerable HTML values;
JS ← extracted vulnerable JS values;
U ← extracted vulnerable URL values;
Temp ← context of each vulnerable string value;
F_API ← Externally available Filtered APIs ($F_1$, $F_2$, $F_3$… $F_N$);
Start
G_TP ← Ø; m ← Ø;
P ← Ø;
C ← Ø;
//generate grouped template for each category of extracted string value
For each $h_i \in$ H do
P ← Levenshtein-distance ($h_i$, $h_{i+1}$);
If (P > $\beta$) then
Accept ($h_i$, $h_{i+1}$);
m ← create grouped template ($h_i$, $h_{i+1}$);
G_TP ← G_TP ∪m;
else
Discard ($h_i$, $h_{i+1}$);
Select other pair;
End for
For each $Js_i \in$ JS do
P ← Levenshtein-distance ($Js_i$, $Js_{i+1}$);
If (P > $\beta$) then
Accept ($Js_i$, $Js_{i+1}$);
m ← create grouped template ($Js_i$, $Js_{i+1}$);
G_TP ← G_TP ∪m;
else
Discard ($Js_i$, $Js_{i+1}$);
Select other pair;
End for
For each $u_i \in$ Udo
P ← Levenshtein-distance ($u_i$, $u_{i+1}$);
If (P > $\beta$) then
Accept ($u_i$, $u_{i+1}$);
m ← create grouped template ($u_i$, $u_{i+1}$);
G_TP ← G_TP ∪m;
else
Discard ($u_i$, $u_{i+1}$);
Select other pair;
End for
// apply filtering API on each template
For each $m_i \in$ G_TP do
C ← context(m($X_i$)) ∈ Temp; //Xi is the placeholder for vulnerable value
$F_i$ ← ($F_i \in$ F_API) && ($F_i \in$ matches C);
Apply $F_i$ to $X_i$;
$H_M$' ← Modify $X_i$ in received response web page;
End for
Return $H_M$';
End
```

Algorithm 4
Stored XSS detection.

```
Input: Server's response (Dres)
Output: Modified response ($D_{res}$')
Att_string ← XSS attack string list;
NScr_list ← Normalized script list ← NULL;
Start
X{}←Ø;
Generate parse tree as P(N, E) ≜ $D_{res}$;
X{}← Vulnerable variable detection and value extraction (P(N, E));
For each $X_i \in$ X{.} do
s ← Normalize($X_i$);
NScr_list ← s' ∪ NScr_list;
End for
For each (s' ∈ NScr_list) do
R ← Boyer-Moore-algorithm(s', Att_string);
If (R) then
XSS indicated;
C ← Identification of Context (X{});
$D_{res}$' ← Grouping and Filtering(C, X{});
return modified $D_{res}$';
else
XSS Free;
return $D_{res}$;
End if
End for
End
```

Algorithm 5
Reflected XSS detection.

```
Input: a) User's command for accessing IoT device ($D_{cmd}$)
b) Server's response ($D_{res}$)
Output: Modified response ($D_{res}$')
Att_string ← XSS attack string list;
cmd_list ← command parameters values ← NULL;
res_list ← response parameters values ← NULL;
Start
For each user's command ($D_{cmd}$) do
$Q_1$ ← getParameterValue($D_{cmd}$);
$Q_1$' ← Decoding($Q_1$);
cmd_list ← cmd_list ∪ $Q_1$';
End for
For each generated server's response ($D_{res}$) do
$Q_2$ ← getParameterValue($D_{res}$);
$Q_2$'← Decoding($Q_2$);
res_list ← res_list ∪ $Q_2$';
End for
For each (x ∈ cmd_list) & (y ∈ res_list) do
R ← Boyer-Moore-algorithm(x, y);
If (R) then
Outcome ← Boyer-Moore-algorithm(x||y, Att_string);
End if
If (outcome) then
XSS indicated;
C ← Identification of Context (cmd_list, res_list);
$D_{res}$' ← Grouping and Filtering(C, res_list, cmd_list);
return modified $D_{res}$';
else
XSS free;
return $D_{res}$;
End if
End for each
End
```

5. Experimental results and performance assessment

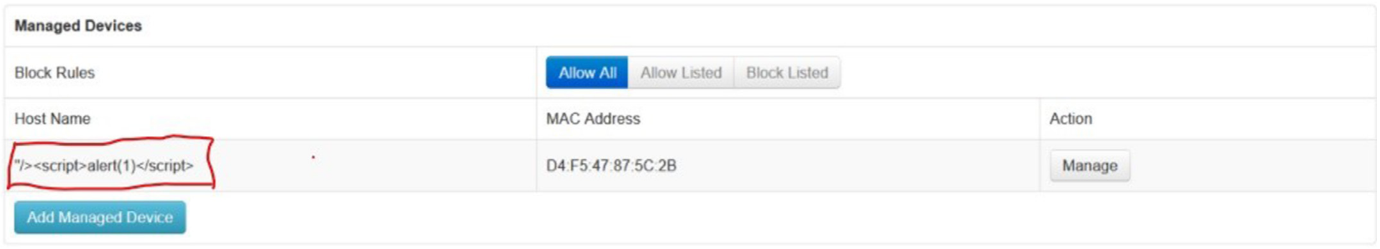
In this section, we demystify the implementation details and observed experimental outcomes of our XSS defensive approach. Later on, we highlight the performance evaluation results and relative study of the proposed approach.

5.1. Implementation details

We have developed the proposed approach by using the Python programming language and integrate it as the extension on the user's browser. For this purpose, we select Intel® Core ™ i5-6600k, 3.9GHz CPU having 16 GB RAM, 1 TB HDD and 256 SSD machine as fog node. It is connected to Azure Cloud service through 32 GB RAM and 2 TB HDD machine. We employ html5lib parser (html5lib parser 2022) to perform parsing of the web page and thus, generates the parsed tree. BeautifulSoup (Beautiful Soup documentation 2022) python library is utilized to traverse the parsed tree that eventually aids in extracting out the useful information. Boyer-Moore algorithm (Boyer and Moore, 1977) is used to com-

Table 2
Relevant encodings category for ("<").

Encoding name	code
URL encoding	%3C
HTML character entity	&lt;
HTML decimal character	<
JS single escape character	\<
JS hex escape sequence	\x3C
HTML hexadecimal character	<
JS Unicode escape sequence	\u003C


Managed Devices

Block Rules | Allow All | Allow Listed | Block Listed

Host Name	MAC Address	Action
"/><script>alert(1)</script>	D4:F5:47:87:5C:2B	Manage

Add Managed Device

Fig. 6. Injected XSS attack string in Managed Device web page.

Table 3
HTML encoding (hexadecimals).

Display	Hexadecimal Code	Numerical Code
“	"	"
#	#	#
&	&	&
‘	’	’
(	(	(
)	)	)
/	/	/
;	;	;
<	<	<
>	>	>

pare the attack payload injected with the XSS attack string black list. It is chosen because it executes fast and produce efficient results when the patterns are relatively lengthier. OWASP and others offer multiple sanitize library (HtmlSanitizer 2022, OWASP Java HTML Sanitizer 2022, DOMPurify 2022) to accomplish malicious script sanitization/filtering task.

5.2. Experimental results

This section elucidates the experimental results and evaluation metrices used to determine the detection efficiency of the proposed approach in the intelligent IoT system. Table 4 highlights the configuration of embedded device which are vulnerable to XSS attack. We have tested two embedded devices against XSS vulnerability that exist in the device management web interface that runs on browser linked to the device's web server. Hitron CODA 4582u contains hidden XSS vulnerability (Hitron CODA 4582u XSS vulnerability 2022) that enables an adversary to inject malicious script code in place of a device name in *managed device* web page by following *wireless->access control->add managed device.* We inject encoded form of simple attack string, for instance, *<script>alert(1)</script>* as *"/><script>alert(1)</script>*, in the vulnerable web page, as shown in Fig. 6. Thus, whenever other user visits the *managed device* web page then injected string triggers the XSS attack, as shown in Fig. 7. Bosch IP camera contains an erroneous URL handler that enables an adversary who possess the information of camera address to craft a malicious URL and sent it to user, thereby, allowing the execution of attack string inside user's browser (Bosch IP camera XSS vulnerability 2022). This is achieved to determine the XSS attack detection efficacy of the proposed approach in the contemporary environment of intelligent IoT application such as Smart Office. In terms of accuracy, we measure percentage of attack strings alleviated by the approach. In terms of performance, we observe the performance overhead related issues like resource required and latency, while executing and integrating the proposed approach on the tested platforms.

To examine the strength or robustness of the proposed approach, we collect a list of attack vectors from five different sources [(Manico and Hansen, 2022, XSS Vectors Cheat Sheet 2022, XSS Script Cheat Sheet for Web Application 2022, Cross-site scripting (XSS) cheat sheet 2022, Heiderich, 2022)] consisting of illicit XSS attack code that can be inserted into web application. We have categorized these attack vectors into six categories on the basis of exploited feature of the web page such as HTML tags, events, JS methods, etc. Table 5 illustrates these attack categories with their example patterns.

We have injected 100 attack strings of each six attack string categories at each vulnerable point in web interface of each tested embedded device. We observe the experimental results of our proposed approach in terms of number of True positive (TP), False Positive (FP), True Negative (TN), False Negative (FN), True Negative Rate (TNR or specificity), False Positive Rate (FPR), False Negative Rate (FNR), and Accuracy. We can calculate these metrices as follows:

$$TNR = \frac{TN}{TN + FP}$$

$$FPR = \frac{FP}{FP + TN}$$

$$FNR = \frac{FN}{FN + TP}$$

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN}$$

Table 6 and Table 7 highlight the observed outcomes of implementing the proposed approach on the web interface of embedded devices. It is observed from Table 6 that the proposed approach attains highest accuracy of 0.98 with lowest FPR value of 0.011 for MJM attack category while testing for router's web interface. On the other hand, we observe the highest accuracy of 0.98 for MJV and MJM attack category while testing for camera's web interface. However, the lowest FPR value is observed for MJM attack category.

Therefore, in both the cases, we noticed that the proposed approach performs best for MJM category and worst for OSEU attack category. For deep attack detection analysis, we have also determined the attack detection rate of our approach on 2 embedded devices web interface platforms.

Table 4
Testing device configuration details.

No.	Device Name	Model No.	Vulnerable firmware version	XSS Vulnerability	Type of XSS attack
1.	Hitron CODA router	4582u	7.1.1.30	CVE-2020-8824	Stored
2.	Bosch IP camera CPP4	Flexidome IP indoor 5000 HD	7.10	CVE-2021-23848	Reflected

Table 5
Six categories of XSS attack strings.

No.	Attack Vector Categories	Example Patterns
1.	Malicious HTML Tags (MHT)	<INPUT TYPE="IMAGE" SRC="javascript:alert('XSS');"> <BODY BACKGROUND="javascript:alert('XSS')"> <BODY ONLOAD=alert('XSS')> <BGSOUND SRC="javascript:alert('XSS');"> <BR SIZE="&{alert('XSS')}"> <TABLE BACKGROUND="javascript:alert('XSS')"> <TABLE><TD BACKGROUND="javascript:alert('XSS')">
2.	Script Embedded Malicious Attributes (SEMA)	<a href = "javascript:document.location='http://www.google.com/'">XSS</A> <a href="http://www.gohttp://www.google.com/ogle.com/">XSS</A> <video src=1 href=1 onerror="javascript:alert(1)"></video> <body src=1 href=1 onerror="javascript:alert(1)"></body> <image src=1 href=1 onerror="javascript:alert(1)"></image>
3.	Exploited HTML Event Method (EHEM)	<IMG SRC= onmouseover="alert('xss')"> <IFRAME SRC=# onmouseover="alert(document.cookie)"></IFRAME> <a onmouseover="alert(document.cookie)">xxs link</a> <a onmouseover=alert(document.cookie)>xxs link</a> <IMG SRC=# onmouseover="alert('xxs')"> <html onMouseOver html onMouseOver="javascript:javascript:alert(1)"></html onMouseOver> <html onMouseEnter html onMouseEnter="javascript:parent.javascript:alert(1)"></html onMouseEnter>
4.	Malicious JS Variable (MJV)	<SCRIPT =">" SRC="http://ha.ckers.org/xss.js"></SCRIPT> <SCRIPT a=">" '' SRC="http://ha.ckers.org/xss.js"></SCRIPT> <SCRIPT "a='>'" SRC="http://ha.ckers.org/xss.js"></SCRIPT> <SCRIPT a='>' SRC="http://ha.ckers.org/xss.js"></SCRIPT> <SCRIPT a=">'>" SRC="http://ha.ckers.org/xss.js"></SCRIPT>
5.	Malicious JS Methods (MJM)	<script>({set/**/$($){_/**/setter=$,_=javascript:alert(1)}}).$=eval</script> <script>{0:#0=eval/#0#/#0#(javascript:alert(1))}</script> <script\x0Atype="text/javascript">javascript:alert(1);</script> '"><\x3Cscript>javascript:alert(1)</script> '"><\x00script>javascript:alert(1)</script> <a href="data:application/x-x509-user-cert;&NewLine;base64&NewLine;,PHNjcmlwdD5hbGVydCgxKTwvc2NyaXB0Pg=="& - #09;& -#10;& - #11>X</a https://www.google<script.com>alert(document.location)</script
6.	Obfuscated Script Embedded URLs (OSEU)	<a href="javas\x01cript:javascript:alert(1)" id="fuzzelement1">test</a> <a href="javas\x05cript:javascript:alert(1)" id="fuzzelement1">test</a> <a href=http://foo.bar/#x='y></a><img alt="'><img src=x:x onerror=javascript:alert(1)></a>"> <div style="list-style:url(http://foo.f)\20url(javascript:javascript:alert(1));">X <META HTTP-EQUIV="refresh" CONTENT="0;url=javascript:javascript:alert(1);"> <META HTTP-EQUIV="refresh" CONTENT="0; URL=http://;URL=javascript:javascript:alert(1);">

Table 6
Observed experimental results of integrating the proposed approach on router's device web interface.

Attack Categories	Total	TP	FP	FN	TN	TNR	FPR	FNR	Accuracy
MHT	100	88	1	3	8	0.889	0.111	0.033	0.96
SEMA	100	85	2	3	10	0.833	0.167	0.034	0.95
EHEM	100	90	1	2	7	0.875	0.125	0.022	0.97
MJV	100	84	2	3	11	0.846	0.154	0.034	0.95
MJM	100	86	1	1	12	0.923	0.077	0.011	0.98
OSEU	100	88	2	5	5	0.714	0.286	0.054	0.93

It is calculated as follows:

$$Attack\ detection\ rate = \frac{TP}{TP+TN+FP+FN}$$

Fig. 8 depicts the detection rate of the approach for all six categories of attack strings on two tested platforms. The overall detection rate for all attack categories in both the cases ranges between 0.8 to 0.9.

5.3. Performance evaluation

This section demystifies the performance assessment of our proposed approach to defend against XSS attack on the device management web interfaces of 2 embedded devices deployed in intelligent IoT system. We have employed two techniques: F-measure and F-Test Hypothesis.

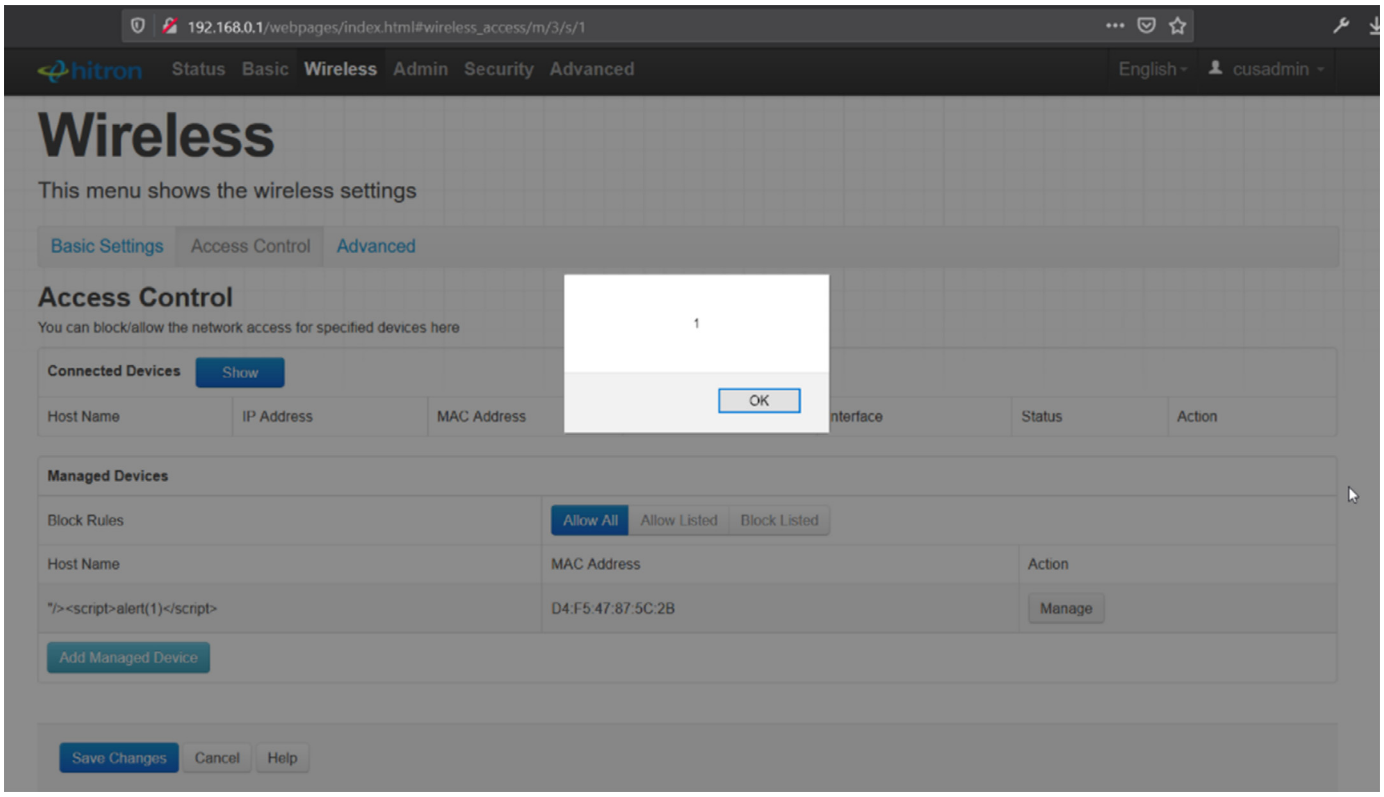
5.3.1. Using F-measure

To evaluate the binary classification problem, precision and recall (TPR or sensitivity) are the evaluation metrics which are used to examine the detection efficiency of the attack detection model. F-measure is the harmonic mean of precision and recall and is considered as an important metric when there is an uneven distribution of class samples within the dataset. These are mathematically calculated as:

$$Precision = \frac{TP}{TP+FP}$$

$$Recall = \frac{TP}{TP+FN}$$

$$F-Measure = \frac{2*Precision*Recall}{Precision+Recall}$$


192.168.0.1/webpages/index.html#wireless_access/m/3/s/1

Wireless

This menu shows the wireless settings

Access Control

You can block/allow the network access for specified devices here

Host Name	MAC Address	Action
"/><script>alert(1)</script>	D4:F5:47:87:5C:2B	Manage

Fig. 7. Execution of injected attack string.

Table 7
Observed experimental results of integrating the proposed approach on Camera's device web interface.

Attack Categories	Total	TP	FP	FN	TN	TNR	FPR	FNR	Accuracy
MHT	100	90	1	2	7	0.875	0.125	0.022	0.97
SEMA	100	88	2	2	8	0.800	0.200	0.022	0.96
EHEM	100	87	2	2	9	0.818	0.182	0.022	0.96
MJV	100	92	1	1	6	0.857	0.143	0.011	0.98
MJM	100	89	1	1	9	0.900	0.100	0.011	0.98
OSEU	100	85	3	1	11	0.786	0.214	0.012	0.96

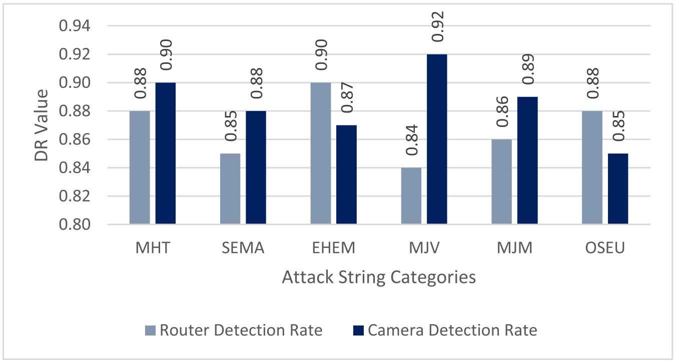


Fig. 8. Attack detection rate on tested platforms.

The conducted performance estimation emphasizes that our proposed approach shows high efficacy since it achieves a high value of F-measure > 0.9 on the tested web application. Table 8 highlights the observed results of detailed performance analysis. It is noticed here that the highest F-measure value of 0.989 is achieved for MJM attack category for both tested platforms.

5.3.2. Using F-test hypothesis

F-test hypothesis is used to identify the variability of observations within the classes under consideration. It supports two hypotheses as follows:

Null Hypothesis: It states that the number of malicious attack strings injected (S_1) is always same as the number of malicious attack strings detected (S_2) i.e., $S_1=S_2$

Alternate Hypothesis: It states that the number of malicious attack strings injected (S_1) is greater than the number of malicious attack strings detected (S_2) i.e., $S_1 > S_2$

We consider significance level ($\alpha = 0.05$). The observed outcomes for f-test are shown in Table 9, Table 10 and Table 11 for both tested devices. In this work, we inject 100 attack strings for each attack string category on both the tested platforms. However, for performance examination by f-test, we inject distinct number of attack strings for each category.

For malicious attack strings injected:

# of attack category (N) = 6

Degree of Freedom DoF (df1) = N − 1 = 5.

For malicious attack strings detected:

# of attack category (N) = 6

Degree of Freedom DoF (df2) = N − 1 = 5.

Table 8
Performance evaluation outcomes of the proposed approach.

Attack Categories	Router Precision	Recall	F-measure	Camera Precision	Recall	F-measure
MHT	0.989	0.967	0.978	0.989	0.978	0.983
SEMA	0.977	0.966	0.971	0.978	0.978	0.978
EHEM	0.989	0.978	0.983	0.978	0.978	0.978
MJV	0.977	0.966	0.971	0.989	0.989	0.989
MJM	0.989	0.989	0.989	0.989	0.989	0.989
OSEU	0.978	0.946	0.962	0.966	0.988	0.977

Table 9
F-test statistical results for attack strings injected on both tested platforms.

#Attack vectors injected (X_i)	(X_i - μ)	(X_i - μ)^2	
90	-2	4	
95	3	9	3.347
97	5	25	
88	-4	16	
91	-1	1	
93	1	1	
Mean (μ) = $\sum X_i/N$= 92		$\sum_1^N (X_i - \mu)^2$= Y= 56	

Table 10
F-test statistical results for attack strings detected on router's web application.

#Attack vectors Detected (X_J)	(X_J - μ)	(X_J - μ)^2	
87	-2	4	
92	3	9	3.130
94	5	25	
86	-3	9	
88	-1	1	
90	1	1	
Mean (μ) = $\sum X_j/N$= 89		$\sum_1^N (X_J - \mu)^2$= W= 49	

Table 11
F-test statistical results for attack strings detected on camera's web application.

#Attack vectors Detected (X_J)	(X_J - μ)	(X_J - μ)^2	
88	-2	4	
92	2	4	3.162
95	5	25	
86	-4	16	
89	-1	1	
90	0	0	
Mean (μ) = $\sum^{X_j}/N$= 90		$\sum_1^N (X_j - \mu)^2$ = W= 50	

Calculate the F-test value ($F_{cal}{}^1$) for router web application, as:

$$\mathbf{F_{cal}{}^1 = S_1{}^2/S_2{}^2 = 11.2024/9.7969 = 1.14345}$$

Calculate the F-test value ($F_{cal}{}^2$) for camera web application, as:

$$\mathbf{F_{cal}{}^2 = S_1{}^2/S_3{}^2 = 11.2024/9.9982 = 1.12044}$$

The tabulated value for f-test (F_{tab}) at α=0.05 and df1=5 and df2=5 is 5.0503. Null hypothesis is rejected if $F_{cal} < F_{tab}$. Since, in our case, $F_{cal}{}^1 < F_{tab}$ and $F_{cal}{}^2 < F_{tab}$ thus, we accept alternate hypothesis that the injected attack strings are greater than strings detected. This is due to some random error while performing the experimentation. We have also calculated the response time of our proposed approach for both types of IoT infrastructure i.e., centralized and distributed (Fog-enabled IoT network). This will assist in analysing the attack detection performance on tested platforms for both IoT structures as many organizations are shifted towards the adoption of fog computing as it provides multiple benefits. Table 12 shows the calculated response time of the proposed approach in both environments.

Table 12
Response time in centralized and distributed IoT structure.

Attack Categories	Router Centralized	Fog-enabled	Camera Centralized	Fog-enabled
MHT	2.09	2.01	2.25	2.18
SEMA	2.31	2.12	2.48	2.32
EHEM	2.78	2.61	2.66	2.54
MJV	2.25	2.12	2.59	2.48
MJM	2.89	2.78	2.85	2.75
OSEU	3.09	3.01	3.12	3.08

5.4. Comparative Analysis

To reflect the robustness and efficiency of our proposed framework, we perform the comparative assessment of our work with the other existing state-of-art techniques including the XSS filters used by the popular web browser like NoScript and CSP for Firefox, NoXSS and XSSAuditor for Chrome.

Table 13 illustrates the comparison on the basis of some significant parameters including Type of XSS Detected (TXD) defines

Table 13
Comparative analysis of the proposed approach.

Parameters → Approaches ↓	Type of XSS Detected	Client-Side Modification	Server-side Modification	Obfuscated Attack String Detection	Partially Injected String Detection	Optimized Sanitization	Nested Context Determination	HTML5 Feature Support
CSP (Content Security Policy 2022)	**All**	×	×	×	×	×	×	√
NoXSS (NoXSS 2022)	**Reflected, DOM**	√	×	×	×	×	×	√
NoScript (Maone, 2022)	**Stored, DOM**	√	×	×	√	×	×	√
XSSAuditor (Bates, Barth, and Jackson, 2010, April)	**Reflected**	×	×	√	×	×	×	×
Gupta et al. (Gupta and Gupta, 2016)	**Reflected, Stored**	×	×	√	×	×	×	×
Duchene et al. (Duchene, Rawat, Richier, and Groz, 2014, March)	**Reflected, Stored**	√	×	×	√	×	×	×
Ahmed et al. (Ahmed and Ali, 2016)	**All**	×	√	×	×	×	×	×
Gupta et al. (Gupta, Chaudhary, and Gupta, 2020)	**Reflected, Stored**	×	√	×	√	×	√	√
Wang et al. (WANG, GU, and ZHAO, 2017)	**Reflected**	√	√	×	√	×	×	×
Our Work	**Stored, Reflected**	×	×	√	√	√	√	√

Listing 1
Dummy Code snippet vulnerable to XSS attack.

```
<html>
<body>
<div name= “abc” onClick= “A()”> look at you!!! </div>
<script>
function A() {
document.getElementByName(“abc”).innerhtml= “Hi” + “$_GET(‘name’)” + “$_GET(‘age’)”;}
</script>
</body></html>
```

the types of XSS attack detected by the method, Client-side Modification (CM) denotes whether the method requires any kind of alteration at client-side or not, Sever-side Modification (SM) means whether the method needs amendments at server-side application or not, Obfuscated Attack String Detection (OASD) signifies whether the technique is capable of identifying encoded or obfuscated scripts or not, Partially Injected Script Detection (PISD) defines whether the approach is capable of detecting partially inserted scripts or not, and Optimized Sanitization (OS) means whether the technique is performing sanitization of the malicious scripts in an optimized fashion or not, Nested Context Determination (NCD) specifies whether the approach determines the accurate context of the attack string or not, HTML5 Feature Support (H5FS) denotes whether the approach is able to identify the attack vectors crafted using the new features of the HTML5 language.

It is perceived from the Table 13 that our proposed approach alleviates two most eminent forms of the XSS attack i.e., Stored and Reflected without any amendments at the client and server-side. It is platform independent framework which is only focused on the communication between IoT device and device server. Therefore, our proposed framework experiences less performance overhead as compared to the existing approaches because it performs optimized filtering of the injected scripts and is applicable to all IoT application domains such as smart home, smart offices, and so on.

5.5. Limitations

Although, the proposed approach is showing good results in the experimental study, nevertheless, there is one drawback of it. Our approach identifies the resemblance between injected attack string and available attack string repository for attack detection. Thus, our work is not able to filtered out the attack strings crafted using new features. If our approach is revised to explore the possibilities of automatically updating the available attack vector repository to handle attack strings crafted using new features, then we can overcome this issue and therefore, it is considered as the future extension of this work. We could also attempt to integrate techniques for automatic validation of the extracted values from the query string.

6. Conclusion

This paper shows that the XSS vulnerability certainly prevails in embedded devices that hampers its security and privacy of the user. XSS may be conducted as a steppingstone to other massive scale cyber-attacks such as DDoS. In this paper, we design an approach to detect and mitigate the harmful effects of the XSS attack. To the best of our information, this is first of its kind research study that specialised in identification and alleviation of XSS attack on embedded devices in intelligent IoT systems. The proposed approach offers protection against stored and reflected XSS attack by comparing the injected attack string with the blacklisted XSS attack vectors. It utilizes the capabilities of the fog computing environment to reduce the latency and bandwidth consumption. To optimize the performance, we perform context-based grouping of the attack strings before applying filtering method. The implementation results shows that our approach is achieving highest accuracy of 0.98. Furthermore, approach's response time is assessed on both fog-enabled IoT system and centralized IoT system. As a part of our future plan, we will explore the possibility of automatically updating the available attack vector repository to handle attack strings crafted using new features.

Response sheet

The authors would like to thank the reviewer for the comments and suggestions, which helped us improve the overall quality of the manuscript. We have gone through the valuable comments/suggestions carefully to improve our manuscript's structure and quality and modified the manuscript accordingly (highlighted in red).

No.	Comments	Explanation	Remarks
Reviewer #1			
1.	Thank you for addressing the comments. The authors have addressed the three main shortcomings of their work. The comparative analysis is very useful, especially Table 13. Also, additions in Section 3.1 provides a nice discussion on the existing gaps on the literature. I think the problem needs a technical editing, also the column headings in Table 13 (although discussed in text) could be represented in a better way, so that the reader can identify what each column represents quickly.	The authors are very thankful to the reviewer for the significant suggestions. As per this comment, we have modified the column headings in Table 13. Although we have technically described the problem in Section 4 by defining each module of the proposed approach, we have checked and corrected it again.	Please refer to Table 13 and Section 4.

Declaration of Competing Interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

CRediT authorship contribution statement

Pooja Chaudhary: Conceptualization, Formal analysis, Writing – original draft. **Brij B. Gupta:** Conceptualization, Writing – review & editing, Supervision. **A.K. Singh:** Conceptualization, Writing – review & editing, Supervision.

References

Html5lib parser [online]. Available at: https://pypi.org/project/html5lib/
NoXSS, 2022. [online]. Available. https://github.com/lwzSoviet/NoXss .
Content Security Policy, 2022. [online] Available https://developer.mozilla.org/en-US/docs/Web/HTTP/CSP .
Cross-site scripting (XSS) cheat sheet, 2022. [online] Available at https://portswigger.net/web-security/cross-site-scripting/cheat-sheet .
XSS Script Cheat Sheet for Web Application, 2022. [online] Available at https://gbhackers.com/top-500-important-xss-cheat-sheet/ .
XSS Vectors Cheat Sheet, 2022. [online] Available at https://gist.github.com/kurobeats/9a613c9ab68914312cbb415134795b45 .
Bosch IP camera XSS vulnerability, 2022. [online] Available at https://nvd.nist.gov/vuln/detail/CVE-2021-23848 .
Hitron CODA 4582u XSS vulnerability, 2022. [online] Available at https://cve.mitre.org/cgi-bin/cvename.cgi?name=2020-8824 .
DOMPurify, 2022. [online]. Available at. https://github.com/cure53/DOMPurify .
OWASP Java HTML Sanitizer, 2022. [online] Available at https://owasp.org/www-project-java-html-sanitizer/ .
HtmlSanitizer, 2022. [online]. Available at. https://github.com/mganss/HtmlSanitizer .
Beautiful Soup documentation, 2022. [online] Available at https://www.crummy.com/software/BeautifulSoup/bs4/doc/ .
Ahmed, M.A., Ali, F., 2016. Multiple-path testing for cross site scripting using genetic algorithms. J. Syst. Archit. 64, 50–62.
Bates, D., Barth, A., Jackson, C., 2010, April. Regular expressions considered harmful in client-side XSS filters. In: Proceedings of the 19th international conference on World wide web, pp. 91–100.
Boyer, R.S., Moore, J.S., 1977. A fast string searching algorithm. Commun. ACM 20 (10), 762–772.
Chaudhary, P., Gupta, B.B., 2018. Plague of cross-site scripting on web applications: a review, taxonomy and challenges. Int. J. Web Based Communities 14 (1), 64–93.
Chaudhary, P., Gupta, S., Gupta, B.B., 2016. Auditing defense against XSS worms in online social network-based web applications. In: Handbook of Research on Modern Cryptographic Solutions for Computer and Cyber Security. IGI Global, pp. 216–245.
Cvitić, I., Peraković, D., Gupta, B., Choo, K.K.R., 2021. Boosting-based DDoS detection in internet of things systems. IEEE Internet of Things J..
Dahiya, A., Gupta, B.B., 2021. A reputation score policy and Bayesian game theory based incentivized mechanism for DDoS attacks mitigation and cyber defense. Future Gener. Computer Syst. 117, 193–204.
Dastjerdi, A.V., Buyya, R., 2016. Fog computing: Helping the Internet of Things realize its potential. Computer 49 (8), 112–116.
Duchene, F., Rawat, S., Richier, J.L., Groz, R., 2014, March. KameleonFuzz: evolutionary fuzzing for black-box XSS detection. In: Proceedings of the 4th ACM conference on Data and application security and privacy, pp. 37–48.
Fang, Y., Li, Y., Liu, L., Huang, C., 2018, March. DeepXSS: Cross site scripting detection based on deep learning. In: Proceedings of the 2018 International Conference on Computing and Artificial Intelligence, pp. 47–51.
Grossman, J., Fogie, S., Hansen, R., Rager, A., Petkov, P.D., 2007. XSS Attacks: Cross Site Scripting Exploits and Defense. Syngress.
Gupta, B.B., Chaudhary, P., Gupta, S., 2020. Designing a XSS defensive framework for web servers deployed in the existing smart city infrastructure. J. Organ. End User Comput. (JOEUC) 32 (4), 85–111.
Gupta, B.B., Li, K.C., Leung, V.C., Psannis, K.E., Yamaguchi, S., 2021. Blockchain-assisted secure fine-grained searchable encryption for a cloud-based healthcare cyber-physical system. IEEE/CAA J. Automatica Sinica 8 (12), 1877–1890.
Gupta, S., Gupta, B.B., 2016. XSS-SAFE: a server-side approach to detect and mitigate cross-site scripting (XSS) attacks in JavaScript code. Arabian J. Sci. Eng. 41 (3), 897–920.
Hameed, S., Khan, F.I., Hameed, B., 2019. Understanding security requirements and challenges in Internet of Things (IoT): A review. J. Comput. Netw. Commun. 2019.
Heiderich, M., 2022. Html5 security cheatsheet [online] Available http://html5sec.org .
Jovanovic, N., Kruegel, C., Kirda, E., 2006, May. Pixy: A static analysis tool for detecting web application vulnerabilities. In: 2006 IEEE Symposium on Security and Privacy (SandP'06). IEEE, pp. 6–pp.
Tewari, A., & Gupta, B. B. (2020). Secure timestamp-based mutual authentication protocol for iot devices using rfid tags. International Journal on Semantic Web and Information Systems (IJSWIS), 16(3), 20–34.
Mani, N., Moh, M., Moh, T.S., 2021. Defending deep learning models against adversarial attacks. Int. J. Software Sci. Comput. Intell. (IJSSCI) 13 (1), 72–89.
Manico, Jim, Hansen, Robert Rsnake, 2022. XSS Filter Evasion Cheat Sheet [online] Available at https://owasp.org/www-community/xss-filter-evasion-cheatsheet .
Maone, Giorgio, 2022. NoScript [online] Available http://www.noscript.net .
Mokbal, F.M.M., Dan, W., Imran, A., Jiuchuan, L., Akhtar, F., Xiaoxi, W., 2019. MLPXSS: an integrated XSS-based attack detection scheme in web applications using multilayer perceptron technique. IEEE Access 7, 100567–100580.
Mokbal, F.M.M., Dan, W., Xiaoxi, W., Wenbin, Z., Lihua, F., 2021. XGBXSS: an extreme gradient boosting detection framework for cross-site scripting attacks based on hybrid feature selection approach and parameters optimization. J. Inform. Secur. Appl. 58, 102813.
Rathore, S., Sharma, P.K., Park, J.H., 2017. XSSClassifier: an efficient XSS attack detection approach based on machine learning classifier on SNSs. J. Inform. Processing Syst. 13 (4), 1014–1028.
Rodríguez, G.E., Torres, J.G., Flores, P., Benavides, D.E., 2020. Cross-site scripting (XSS) attacks and mitigation: A survey. Computer Networks 166, 106960.
Salas, M.I.P., Martins, E., 2014. Security testing methodology for vulnerabilities detection of xss in web services and ws-security. Electron. Notes Theor. Comput. Sci. 302, 133–154.
Salhi, D.E., Tari, A., Kechadi, M.T., 2021. Using clustering for forensics analysis on internet of things. Int. J. Software Sci. Comput. Intell. (IJSSCI) 13 (1), 56–71.
WANG, D., GU, M., ZHAO, W., 2017. Cross-site script vulnerability penetration testing technology. J. Harbin Eng. Univers. 38 (11), 1769–1774.
Zhang, X., Zhou, Y., Pei, S., Zhuge, J., Chen, J., 2020. Adversarial examples detection for XSS attacks based on generative adversarial networks. IEEE Access 8, 10989–10996.
Zhou, Y., Wang, P., 2019. An ensemble learning approach for XSS attack detection with domain knowledge and threat intelligence. Comput. Security 82, 261–269.

Pooja Chaudhary is currently pursuing her Ph.D. degree from National Institute of Technology Kurukshetra, Haryana, India. She has completed her M.Tech in In formation and Cyber security from the same institute. She has published her research work in many journals and conference proceedings of IEEE, Springer, Elsevier, and Taylor and Francis. Her area of interest includes IoT security, Online Social Network security, fog computing, Information-security.

Brij B. Gupta received the PhD degree from Indian Institute of Technology (IIT) Roorkee, India. In more than 16 years of his professional experience, he published over 400 papers in journals/conferences including 25 books and 05 Patents with over 13800 citations. He has received numerous national and international awards including Canadian Commonwealth Scholarship (2009), Faculty Research Fellowship Award (2017), from the Govt. of Canada, MeitY, GoI, IEEE GCCE outstanding and WIE paper awards and Best Faculty Award (2018 & 2019), NIT Kurukshetra, respectively. He is also selected in the 2021 and 2020 Stanford Universitys ranking of the worlds top 2% scientists. He is/was also a visiting/adjunct professor with several universities worldwide. He is also an IEEE Senior Member (2017) and also selected as 2021 Distinguished Lecturer in IEEE CTSoc. Dr Gupta is also serving as Member-in-Large, Board of Governors, IEEE Consumer Technology Society (2022-204). Prof. Gupta is also leading IJSWIS, IJSSCI and IJCAC, IGI Global, as Editor-in-Chief. Moreover, he is also serving as lead-editor of a Book Series with CRC and IET press. He also served as TPC members in more than 150 international conferences. Dr Gupta is also serving/served as Associate/Guest Editor of various journals and transactions. At present, Prof. Gupta is working as Professor with the Department of Computer Science and Information Engineering (CSIE), Asia University, Taichung, Taiwan. His research interests include information security, Cyber physical systems, cloud computing, blockchain technologies, intrusion detection, AI, social media and networking.

Awadhesh Kumar Singh received his MTech and Ph.D. Engineering degrees in Computer Science from Jadavpur University, Kolkata, India, in 1998 and 2004, respectively, and BTech degree in Computer Science from Madan Mohan Malaviya University of Technology, Gorakhpur, India, in 1988. His research areas are cognitive radio networks, security, fault tolerance, and distributed algorithms. Presently he is serving as a professor in the Department of Computer Engineering at the National Institute of Technology, Kurukshetra, India. He also served as head of the Computer Engineering Department during 2007-2009 and 2014-2016.