

Comparing the Detection of XSS Vulnerabilities in Node.js and a Multi-tier JavaScript-based Language via Deep Learning

Héloïse Maurel¹, Santiago Vidal²^a and Tamara Rezk¹^b

¹INRIA, INDES Project, Sophia Antipolis, France

²ISISTAN-CONICET, Tandil, Argentina

heloise.maurel@inria.fr; santiago.vidal@isistan.unicen.edu.ar; tamara.rezk@inria.fr

Keywords: Web security, Deep learning, Web attacks, Cross-Site Scripting

Abstract: Cross-site Scripting (XSS) is one of the most common and impactful software vulnerabilities (ranked second in the CWE 's top 25 in 2021). Several approaches have focused on automatically detecting software vulnerabilities through machine learning models. To build a model, it is necessary to have a dataset of vulnerable and non-vulnerable examples and to represent the source code in a computer understandable way. In this work, we explore the impact of predicting XSS using representations based on single-tier and multi-tier languages. We built 144 models trained on Javascript-based multitier code - i.e. which includes server code and HTML, Javascript and CSS as client code - and 144 models trained on single-tier code, which include sever code and client-side code as text. Despite the lower precision, our results show a better recall with multitier languages than a single-tier language, implying an insignificant impact on XSS detectors based on deep learning.

1 Introduction


Web injection vulnerabilities on the client side, a.k.a. cross-site scripting or XSS, are pervasive and have been on top-ranked vulnerability lists for over 10 years. XSS vulnerabilities are caused by a flow of information, coming from untrusted input, to a sensitive sink. This flow of information usually follows a path from the client to the sever and back to (possibly other) clients. In order to prevent XSS vulnerabilities it is enough to place sanitizers, which are adapted to the context of the sink. However, placing sanitizers is tricky and error-prone which justifies the large existing body of woks studying the problem of XSS detection and prevention as for example (Luo et al., 2011; Somé et al., 2016; Doupé et al., 2010; Schoepe et al., 2016; Melicher et al., 2018; Livshits and Chong, 2013; Lekies et al., 2017; Balzarotti et al., ; Gundy and Chen, 2009; Staicu et al., 2018). In particular, previous works have also studied how well deep learning techniques can help detect this kind of vulnerabilities (Maurel et al., 2021; Melicher et al., 2021; Fang et al., 2018; Chen et al., 2019; Mokbal et al., 2019; Abaimov and Bianchi, 2019; Shar and Tan, 2013). Our focus here is on static detection of


XSS vulnerabilities when the flows from sources to sinks flow via the server (known as XSS of the first and second type or reflected and stored XSS) and source code from the server is available. In recent years several techniques for code representations for deep learning have arised (Alon et al., 2019; Shar and Tan, 2013; Li et al., 2018a; Li et al., 2018b; Russell et al., 2018). We are interested here in code representation techniques based on programming languages processing or PLP (Alon et al., 2019) and the influence of more expressive abstract syntax trees in order to detect XSS in web applications.

Traditionally, web applications execute in several tiers including the client tier and the server tier. To implement these tiers, developers need different languages - e.g. Javascript for the web client and PHP or Node.js for the web server.

Multi-tier programming (Serrano et al., 2006), (Cooper et al., 2006) is a programming paradigm for distributed software that has arised in 2006 in order to simplify the programming task and use a single language to program all the tiers. This language homogenization offers several advantages concerning development, maintenance, scalability, and analysis of web applications (Weisenburger et al., 2020).

Previous work (Maurel et al., 2021) obtained significant results to identify XSS using deep learning comparing different code representation techniques

^a <https://orcid.org/0000-0003-2440-3034>

^b <https://orcid.org/00000-0003-3744-0248>

based on NLP and PLP for PHP and Node.js and observed a difference in impact by including client-side content in the form of text or code in the learning process of NLP techniques. However, because of the need of building AST representations in the pre-processing step and the absence of an appropriate parser to build models for including the HTML, JavaScript and CSS as code for PHP and Node.js, they could not evaluate the PLP approach that they used. In this work, we fill this gap by studying the impact of including client-side code as text or code (using the more expressive multitier ASTs) in learning such vulnerabilities detectors by comparing Node.js and the multitier language Hop.js (Serrano, 2006; Serrano and Prunet, 2016) using the PLP approach.

Contributions. In summary, our contributions are:

- We build a new generator for Hop.js, a multitier language based on JavaScript and datasets for Hop.js classified as XSS secure or insecure (Section 3).
- We propose a new XSS static analyzer for Hop.js based on deep learning and the PLP code representation technique (Section 4).
- We evaluate models in two different datasets one including HTML/JavaScript/CSS as code in Hop.js and one including it as text in Node.js, using PLP as code representation for deep mode languages. Finally, we compare our results (Section 5).

2 Hop.js and Node.js Languages

For our experiments we have chosen two languages to program web applications which are based on JavaScript: Hop.js (Serrano and Prunet, 2016) and Node.js (Node.js, 2021). Hop.js (Serrano and Prunet, 2016) is a multitier language³ based on the JavaScript language and it is the successor of one of the two first multitier languages that existed HOP (Serrano, 2006).

Figure 1 shows a “hello world” multi-tier web application in Hop.js with the special HopScript service declaration statement. HopScript services, as shown in line 1, are distinguished from regular Javascript functions by using the `service` keyword. In this way, the `server` function in line 1 is a Javascript remotely callable function via HTTP protocol.

Figure 2 shows the same “hello world” application but written in Node.js for the server-side. As it is

```

1 'use hopscript';
2 service server() {
3   return <html>
4     <body>
5       <h1> Hello World </h1>
6     </body>
7   </html> ;
8 }

```

Figure 1: Hop.js sample - HTML markup included in the Javascript syntax as code.

```

1 let http= require( 'http' );
2 let server= http.createServer(
3   function(req, res){
4     res.write( "<html> <body> <h1>Hello World!</h1> </body> </html>" );
5     res.end() ;
6   } );
7 server.listen(8080);

```

Figure 2: Node.js sample - HTML markup included as.

shown, Node.js describe the client-side code in plain-text inside quotes. In contrast, in Figure 1, Hop.js embeds client-side expression using Hop.js functions that look similar to HTML markup containers.

If we represent the AST of the previous examples, we can notice that, for Hop.js (Figure 3b), the client-side structures can be extracted and parsed by using its AST. On the contrary, for Node.js (Figure 3a), these structures are only represented as a string value and therefore cannot be parsed. Thus, the Hop.js AST is more expressive than the one on Node.js. It is essential when an analysis needs to extract information from the AST. For example, if a classifier algorithm wants to be a model for XSS prediction, an AST built for Hop.js will likely include more information to extract than an AST built from Node.js.

3 Hop.js Database for XSS

In this work, we compare the effect of detecting XSS vulnerabilities in a multi-tier language based JavaScript language and Node.js with deep learning models. With that goal in mind, we represent the source code using ASTs. AST structures are widely used in the pre-processing stages of programming languages to analyze code at different granularity such as declaration level (Shar and Tan, 2013), function level (Lin et al., 2018), the intra-procedural level (Li and Zhou, 2005) and the file level (Wang et al., 2016; Dam et al., 2017).

To build deep learning models that detects XSS, having a large ground-truth database of secure and insecure source code is one of the major obstacles. Only a few works have constructed real-world datasets for evaluation. However, these datasets are generally small, providing insufficiently labelled vulnerability

³<http://hop.inria.fr>

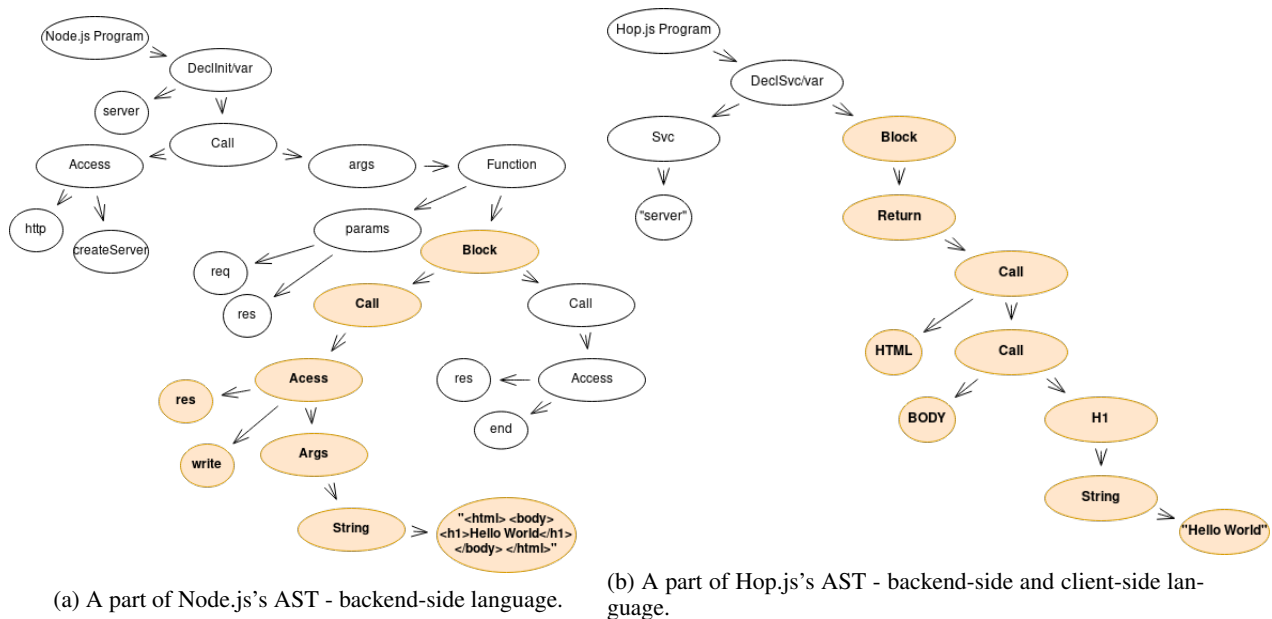


Figure 3: Comparison of single-tier programming and multi-tier programming by the informations obtained in analyzing the code in Figure 2 and 1.

data (Lin et al., 2019; Lin et al., 2018), offering synthetic samples that cannot be compiled (Choi et al., 2017; Sestili et al., 2018) or are not publicly available.

Another challenge related to creating a large ground truth database is the need to label every sample of the real-world’s datasets. As it stands, this is tedious work done by hand for most research work (Shar and Tan, 2013; Lin et al., 2018; Li and Zhou, 2005; Li et al., 2018b).

Additionally, we need comparable web applications to assess the influence of semantic knowledge transcribed via AST for stand-alone multi-tier and single-tier web applications. For all of these reasons, in this work, we build a synthetic database that could be used as a benchmark dataset. For comparing this dataset with Node.js, we use the Node.js generator created by previous work (Maurel et al., 2021).

Because supervised deep learning requires to label, in our case, as secure or insecure each sample of the database, our generator is based on the OWASP XSS cheatsheet series project (OWASP, 2021). This project proposes a positive model of rules using output encoding or filtering to prevent XSS attacks. We present the implementation of the OWASP rules in Hop.js in Section 3.2.

3.1 Hop.js Generator

We implement a synthetic generator in Hop.js.js mixing server-side and client-side sources for XSS vul-

nerabilities. We generated 34,400 standalone Web applications (33 LOC on average).

Two main components constitute this generator. First, the generation of the samples itself. The Hop.js generator combines 16 user inputs, 84 incorrect and proper sanitisations, and 25 construction templates that follow OWASP rules (OWASP, 2021). The second component is a classification system of samples. This system classifies samples as secure or insecure.

The generator aggregates four code snippets to produce a single sample. First, the start of a sample begins and ends respectively - depending on the available build templates - with start and end build fragments. Second, the generator chooses one possible Hop.js user input fragment and insert it between the beginning and the end of the construction fragments. Third, the generator gives - to the sample - a proper, improper or no sanitisation to try to prevent XSS or not. This type of sanitization follows the input fragment. Finally, the classifier labels the sample as secure or insecure depending on the sanitization chosen by the generator and the HTML context of the sink.

A sample is considered insecure for XSS when there is a flow between a source and a sink, without use of an appropriate sanitizer. A source is the entry-point of user inputs where a malicious user can eventually inject a payload. Listing 1 shows a part of a generator input where the value of the `userData` parameter can be a malicious payload injection point.

```
1 let urlVar= require('url');
```

```

2 let untrustedVar= urlVar.parse(this.path, true).
  query.userData;

```

Listing 1: Read the `userData` field from the server URL query parameter when an HTTP GET request method is called.

A sink renders the linked source on the web application and can potentially execute its malicious content. Our generator uses 25 different sinks and they are part of the construction templates - see Section 3.2 for more details.

The Hop.js generator is able to generate samples with sanitized flows, unsanitized flows, incorrectly sanitized flows and malformed flows between sources and sinks. In the case of an incorrectly sanitized flow, the generator can define a proper sanitization but without applying it to the flow.

The classification component is based on the encoding and filtering recommendations giving by the OWASP rules (OWASP, 2021). Depending on the context of the sink, the potential link between the source and the sink, and the type of sanitization used in a sample, the classifier can label the sample as secure or insecure. In this way, we generate 18,624 secure Hop.js samples and 15,776 insecure Hop.js samples.

3.2 OWASP Rules Implementation in Hop.js

The OWASP XSS cheatsheet series project (OWASP, 2021) proposes rules using output encoding or filtering to prevent XSS attacks.

In this section, we introduce the implementation in Hop.js of the first six OWASP rules - that are used by our generator. The last two rules relate to javascript URL avoidance and DOM-XSS prevention. Avoiding javascript URLs does not help us generate unsafe samples and DOM-XSS recently has its own OWASP rules which will be an extension option for future work.

The whole listings described in each part of this section used two Hop.js notations - ``${}`` and ``${}`` - and two variables defined in Figure 4 : `head_var` and `body_var`. The `head_var` variable contains all the HTML code needed to describe the header of any HTML web application. In the case of `body_var`, this variable contains all the client-side code describing all the content and HTML structure of the web application.

The ``${}`` and ``${}`` notations are applied to indicate, at compilation-time, which part belongs to server-side code and which part belongs to client-side code.

```

1 let headVar= <head>
2 <title>Web App's name</title>
3 </head>;
4 let bodyVar= <body>
5 <h1>All the content of the Web application</h1>
6 <p>...</p>
7 </body>;

```

Figure 4: `head_var` and `body_var` definition used in the whole listing examples.

3.2.1 Rule #0 - Never Insert Untrusted Data Except in Allowed Locations

OWASP recommends that developers never put untrusted data directly into five HTML contexts. We implement these contexts for our Hop.js dataset.

First, developers could insert an unreliable user data value - contained in a variable `untrustedVar` - in a HTML attribute name.

In the following code, the variable `untrustedVar` is an attribute of a HTML `<div>` tag:

```

1 let divVar= '<div ' + untrustedVar + ' = "a" />';
2 bodyVar.appendChild(divVar);
3 return <html> ${headVar} ${bodyVar} </html>;

```

The following is an full percent-encoded URL exploit for the above vulnerable code:

```

1 %3E%3c%73%63%72%69%70%74%3e%61%6c%65%72%74%28%31%29%3c
2 %2f%73%63%72%69%70%74%3e
3 //Decoded version
4 <script>alert(1)<%2fscript>

```

Second, developers have the possibility to insert an untrusted user data value - contained in a `untrustedVar` variable - in a HTML comment.

In the following code, the `untrustedVar` variable is inside an HTML `<!--` tag. This untrusted comment will become part of the Web application's metadata. Since metadata is not displayed in HTML `<head>` tag and the untrusted data is inside a comment, it will be hidden from the user.

```

1 commentVar = '<!-- ' + untrustedVar + ' -->';
2 headVar.appendChild(commentVar);
3 return <html> ${headVar} ${bodyVar} </html>;

```

The following is an full percent-encoded URL exploit for the above vulnerable code:

```

1 %2D%2D%3E%3c%73%63%72%69%70%74%3e%61%6c%65%72%74%28
2 %31%29%3c%2f%73%63%72%69%70%74%3e%3C%21%2D%2D
3 //Decoded version
4 --<script>alert(1)<%2fscript><!--

```

Third, developers have the option of using an untrusted user data value - contained in a `untrustedVar` variable - inside a complex javascript function.

In the following piece of code, the `untrustedVar` variable is called inside the body of the function called `foo`. This method will be called from the client-side when the `<body>` element has finished loading into the browser.

```

1 let scriptVar= <script/>;
2 let funcVar= 'function foo(){';
3 funcVar= funcVar+ untrustedVar+ ''';
4 scriptVar.appendChild(funcVar);
5 let bodyVar= <body onload="{foo()}>
6 <h1> Hello World! </h1>
7 </body>;
8 return <html> ${headVar}
9 ${scriptVar} ${bodyVar}
10 </html>;

```

The following is an exploit for the above vulnerable code (document.vulnerable contains the boolean type true encoded with an esoteric programming style):

```

1 document.vulnerable=!![];return alert(
document.vulnerable);

```

Fourth, developers have the option of using an untrusted user data value - contained in a untrustedVar variable - inside <script> element.

In the following piece of code, the untrustedVar variable is an expression of the HTML <script> tag - which is inserted inside the HTML structure of the web application. In this context, untrustedVar variable could contain any malicious javascript script and this script will be executed by the browser.

```

1 let scriptVar = <script/>;
2 scriptVar.appendChild(untrustedVar);
3 return <html> ${headVar}
4 ${scriptVar} ${bodyVar} </html>;

```

The following is an full percent-encoded URL exploit for the above vulnerable code:

```

1 %6D%61%6C%69%63%6F%75%73%31%3D%70%72%6F%6D%70%74%28%22
2 %70%61%73%73%77%6F%72%64%22%29%3B%6D%61%6C%69%63%6F%75
3 %73%32%3D%70%72%6F%6D%70%74%28%22%6C%6F%67%69%6E%22%29
4 %3B%63%6F%6E%73%6F%6C%65%2E%6C%6F%67%28%6D%61%6C%69%63
5 %6F%75%73%31%29%3B%63%6F%6E%73%6F%6C%65%2E%6C%6F%67%28
6 %6D%61%6C%69%63%6F%75%73%32%29%3B
7 //Decoded version
8 malicious1=prompt("password");
9 malicious2=prompt("login");console.log(malicious1);
10 console.log(malicious2);

```

Fifth, developers have the option of using an untrusted user data value - contained in a untrustedVar variable - inside <style> style sheet informations.

In the following piece of code, the untrustedVar variable is an expression of the CSS <style> tag - which is inserted inside the HTML structure of the web application. In this context, untrustedVar variable could force the execution of any malicious javascript script and this script will be executed by the browser.

```

1 let styleVar = <style/>;
2 styleVar.appendChild(untrustedVar);
3 return <html> ${headVar} ${styleVar}
4 ${bodyVar} </html>;

```

The following is an full percent-encoded URL exploit for the above vulnerable code:

```

1 </style><script>age = prompt('How old are you?', 101);
2 alert(`data user ${age}`);</script>

```

Finally, developers can use an untrusted user data value - contained in a untrustedVar variable - to create a custom HTML tag name.

In the following piece of code, the untrustedVar variable is a tag name that can help structure the content of the web application. In this context, untrustedVar variable could force the execution of any malicious javascript script and this script will be executed by the browser.

```

1 let tag_var = '<' + untrustedVar + 'href= "/bob" />';
2 bodyVar.appendChild( tag_var );
3 return <html> ${headVar} ${bodyVar} </html>;

```

The following is an full percent-encoded URL exploit for the above vulnerable code:

```

1 %73%63%72%69%70%74%3E%20%61%6C%65%72%74%28%29%3C%2F
2 %73%63%72%69%70%74%3E
3 //Decoded version
4 script>alert(2)</script>

```

3.2.2 Rule #1 - HTML Encode Before Inserting Untrusted Data into HTML Element Content

In this rule, OWASP recommends that developers encode HTML before inserting it into HTML content. To illustrate this point, OWASP gives two use cases and we implemented them with Hop.js.

First, developers can use an untrusted user value - contained in a variable untrustedVar - inside the content of the structural HTML tag such a <div>.

In the following code, the variable untrustedVar is inserted inside the content of the HTML <div> tag:

```

1 let divVar = <div/>;
2 divVar.appendChild(untrustedVar);
3 bodyVar.appendChild(divVar);
4 return <html> ${headVar} ${bodyVar} </html>;

```

The following is an exploit for the above vulnerable code:

```
let untrustedVar= "><script>alert(`XSS`)</script>";
```

Last, developers can apply an untrusted user data value - contained in a untrustedVar variable - inside the structural body of HTML.

In the following piece of code, the untrustedVar variable is inside the web application's <body> markup.

```

1 bodyVar.appendChild(untrustedVar);
2 return <html> ${headVar} ${bodyVar} </html>;

```

The following is an full percent-encoded URL exploit for the above vulnerable code:

```

1 %3C%73%74%79%6C%65%20%6F%6E%6C%6F%61%64%3D%22%61%6C%65
2 %72%74%281%29%22%3E%6D%61%6C%69%63%6F%75%73%3C%2F%73
3 %74%79%6C%65%3E
4 //Decoded version
5 <style onload="alert(1)">malicious</style>

```

3.2.3 Rule #2 - Attribute Encode Before Inserting Untrusted Data into HTML Common Attributes

In this rule, OWASP recommends that developers encode untrusted attribute values before inserting them into HTML common attributes. To illustrate this point, OWASP gives one use case. Derived from it, we implemented three HTML contexts in Hop.js.

First, developers can use an untrusted user data value - contained in a variable `untrustedVar` - to define an unquoted value of common attributes.

In the following piece of code, the variable `untrustedVar` is the unquoted value of a `<div>` tag attribute. Unquoted values can be interrupted by many characters, unlike simple quote or double quote values.

```
1 let divVar = '<div id=' + untrustedVar + '>content</div>';
2 bodyVar.appendChild(divVar);
3 return <html> ${headVar} ${bodyVar} </html>;
```

The following is an full encoding URL exploit for the above vulnerable code:

```
1 %74%77%6F%78%73%73%20%6F%6E%63%6C%69%63%6B%3D%22%61%6C
2 %65%72%74%28%29%22%3E%3C%73%63%72%69%70%74%3E%61%6C
3 %65%72%74%28%22%78%73%73%22%29%3C%2F%73%63%72%69%70%74
4 //Decoding version
5 twoxss+onclick="alert(1)"><script>alert("xss")</script
```

Second, developers can use an untrusted user data value - contained in a variable `untrustedVar` - to define a simple quote value of common attributes.

In the following piece of code, the variable `untrustedVar` is included inside simple quote of a `<div>` tag attribute. Simple quote ' character can be only interrupted by the corresponding simple quote '.

```
1 let divVar = "<div id='" + untrustedVar + "'>content</div>";
2 bodyVar.appendChild(divVar);
3 return <html> ${headVar} ${bodyVar} </html>;
```

The following is an exploit for the above vulnerable code:

```
let untrustedVar= "'><script>alert(\"XSS\")</script>";
```

Last, developers can use an untrusted user data value - contained in a variable `untrustedVar` - to define a double quote value of common attributes.

In the following piece of code, the variable `untrustedVar` is included inside double quote of a `<div>` tag attribute. Double quote " character can be only interrupted by the corresponding double quote " unlike unquoted.

```
1 let divVar = '<div id="' + untrustedVar + '">content</div>';
2 bodyVar.appendChild(divVar);
3 return <html>${headVar} ${bodyVar}
4 </html>;
```

The following is an exploit for the above vulnerable code:

```
let untrustedVar= '"><script>alert(\"XSS\")</script>';
```

3.2.4 Rule #3 - JavaScript Encode Before Inserting Untrusted Data into JavaScript Data Values

OWASP advises developers to place untrusted data only inside quoted data values in JavaScript code. To illustrate this point, OWASP gives four use cases. and we implemented them with Hop.js. Derived from these use cases, we implemented seven HTML contexts in Hop.js.

First, developers can use an untrusted user data value - contained in a variable `untrustedVar` - inside simple quoted event handler values.

In the following piece of code, the variable `untrustedVar` is inserted inside simple quoted of a `onmouseover` event.

```
1 bodyVar.appendChild("<div onmouseover= \"x=\" + untrustedVar + \"'\>\" );
2 return <html> ${headVar} ${bodyVar} </html>;
```

The following is an full encoding URL exploit for the above vulnerable code:

```
1 %78%73%73%27%22%3E%3C%73%63%72%69%70%74%3E%6D%61%6C%69
2 %63%69%6F%75%73%3D%70%72%6F%6D%70%74%28%22%70%61%73
3 %73%77%6F%72%64%22%29%3B%6D%61%6C%69%63%69%6F%75%732
4 %3D%70%72%6F%6D%70%74%28%22%29%3B%61%6C%65%72
5 %74%28%6D%61%6C%69%63%69%6F%75%73%2B%22%3A%22%2B%6D
6 %61%6C%69%63%69%6F%75%732%29%3B%3C%2F%73%63%72%69%70
7 %74%3E
8 //Decoded version
9 xss'"><script>malicious1=prompt("password");
10 malicious2=prompt("login");
11 alert(malicious1%2B":"%2Bmalicious2);</script>
```

Second, developers can use an untrusted user data value - contained in a variable `untrustedVar` - inside double quoted event handler values.

In the following piece of code, the variable `untrustedVar` is inserted inside double quoted of a `onmouseover` event.

```
1 bodyVar.appendChild("<div onmouseover= \"x=\"+ untrustedVar+\"'\>\" );
2 return <html> ${headVar} ${bodyVar} </html>;
```

The following is an exploit for the above vulnerable code:

```
1"onclick=alert("xss")>click</div>
```

Third, developers can use an untrusted user data value - contained in a variable `untrustedVar` - inside simple quoted string and used in JavaScript script.

In the following piece of code, the variable `untrustedVar` is inserted inside simple quoted of an alert box.

```
1 let scriptVar = <script/>;
2 scriptVar.appendChild("alert('"+ untrustedVar + "')");
3 return <html>${headVar} ${scriptVar} ${bodyVar}</html>
```

The following is an exploit for the above vulnerable code:

```
normal')</script><script>alert("xss")</script><script>
```

Fourth, developers can use an untrusted user data value - contained in a variable `untrustedVar` - inside double quoted string and used in JavaScript script.

In the following piece of code, the variable `untrustedVar` is inserted inside double quoted of an alert box.

```
1 let scriptVar = <script/>;
2 scriptVar.appendChild("alert (\\""+untrustedVar+"\")");
3 return <html>${headVar} ${scriptVar} ${bodyVar}</html>
```

The following is an exploit for the above vulnerable code:

```
1 normal"></script><button onafterscriptexecute=
2 alert(1)><script>1</script>
```

Fifth, developers can use an untrusted user data value - contained in a variable `untrustedVar` - inside JavaScript simple quoted assignments.

In the following piece of code, the variable `untrustedVar` is inserted between simple quoted to define the value of the variable `x`.

```
1 let scriptVar = <script/>;
2 scriptVar.appendChild( "x= '" + untrustedVar + "' );
3 return <html>${headVar} ${scriptVar} ${bodyVar}</html>
```

The following is an exploit for the above vulnerable code:

```
1 3';data_user=prompt("First Name");
2 alert(data_user);y=3
```

Sixth, developers can use an untrusted user data value - contained in a variable `untrustedVar` - inside JavaScript double quoted assignments.

In the following piece of code, the variable `untrustedVar` is inserted between double quoted to define the value of the variable `x`.

```
1 let scriptVar = <script/>;
2 scriptVar.appendChild( "x= \"" + untrustedVar + "\"";
3 return <html>${headVar} ${scriptVar} ${bodyVar}</html>
```

The following is an exploit for the above vulnerable code:

```
1 xss";data_user=prompt("xss");y=data_user;
2 console.log(y);z="xss
```

Finally, OWASP warns to never use unreliable data as input to built-in javascript functions such as `setInterval`. However, any developer can use the following format:

```
1 let scriptVar = <script/>;
2 scriptVar.appendChild( "window.setInterval( '" +
  untrustedVar + "' );" );
3 return <html> ${headVar} ${scriptVar} ${bodyVar} </
  html>
```

The following is an exploit for the above vulnerable code:

```
1 console.log("xss3");',1000);
2 setTimeout("console.log('xss2');", 500);alert('xss1
```

3.2.5 Rule #4 - CSS Encode And Strictly Validate Before Inserting Untrusted Data into HTML Style Property Values

OWASP advices developers to place untrusted data only inside property value in CSS style. To illustrate this point, OWASP gives three use cases. Derived from these use cases, we implemented four HTML contexts in Hop.js.

First, developers can use an untrusted user data value - contained in a variable `untrustedVar` - inside double quoted CSS property value.

In the following piece of code, the variable `untrustedVar` is inserted inside double quoted of a `color` property.

```
1 let style_var = <style/>;
2 style_var.appendChild("body { color : \"" +
  untrustedVar + "\";");
3 bodyVar.appendChild( style_var );
4 return <html> ${headVar} ${bodyVar} </html>;
```

The following is an exploit for the above vulnerable code:

```
1 ;}@keyframes x{}</style>
2 <xss style="animation-name:x" onanimationend=
3 "alert('xss')"></xss>
```

Second, developers can use an untrusted user data value - contained in a variable `untrustedVar` - inside simple quoted CSS property value.

In the following piece of code, the variable `untrustedVar` is inserted inside simple quoted of a `color` property.

```
1 let style_var = <style/>;
2 style_var = style_var + "body { color : '" +
  untrustedVar + "';" + '</style>';
3 bodyVar.appendChild( style_var );
4 return <html> ${headVar} ${bodyVar} </html>;
```

The following is an exploit for the above vulnerable code:

```
1 '})@keyframes xss{}</style>
2 <img style="animation-name:xss"+
3 onwebkitanimationend='alert("xss")'></img>
```

Third, developers can use an untrusted user data value - contained in a variable `untrustedVar` - inside an unquoted CSS property value.

In the following piece of code, the variable `untrustedVar` is inserted as an unquoted value of a `color` property.

```
1 let style_var = <style/>;
2 style_var = style_var + "body { color : " +
  untrustedVar + ";" + '</style>';
3 bodyVar.appendChild( style_var );
4 return <html> ${headVar} ${bodyVar} </html>;
```

The following is an exploit for the above vulnerable code:

```
1 blue;</style><script>alert("xss")</script>
```

Finally, developers can use an untrusted user data value - contained in a variable `untrustedVar` - inside a style attribute value of HTML markup.

In the following piece of code, the variable `untrustedVar` is inserted inside a style value of a HTML `` markup.

```
1 bodyVar.appendChild("<span style = \"color :\" +
  untrustedVar + \"\> Hey </span>");
2 return <html> ${headVar} ${bodyVar} </html>;
```

The following is an exploit for the above vulnerable code:

```
blue"+onclick=alert(document.cookie);+b="
```

3.2.6 Rule #5 - URL Encode Before Inserting Untrusted Data into HTML URL Parameter Values

OWASP warns developers to encode untrusted data before to put it inside HTTP GET parameter values. To illustrate this point, OWASP gives one use case. Derived from it, we implemented three HTML contexts in Hop.js.

First, developers can use an untrusted user data value - contained in a variable `untrustedVar` - in double quoted hyperlink attribute values.

In the following code, the variable `untrustedVar` is inserted inside double quoted of an HTML `href` attribute value:

```
1 bodyVar.appendChild("<a href=\""+untrustedVar+"\">
2 link</a>");
3 return <html> ${headVar} ${bodyVar} </html>;
```

The following is an exploit for the above vulnerable code:

```
1 javascript:document.vulnerable=true;
2 alert(document.vulnerable);
```

Second, developers can use an untrusted user data value - contained in a variable `untrustedVar` - in simple quoted hyperlink attribute values.

In the following piece of code, the variable `untrustedVar` is inserted inside simple quoted of an HTML `href` attribute value:

```
1 bodyVar.appendChild("<a href='"+untrustedVar+"'>
2 link</a>");
3 return <html> ${headVar} ${bodyVar} </html>;
```

The following is an exploit for the above vulnerable code - the variable `x` contains a true boolean value encoded with an esoteric programming style:

```
1 javascript:javascript:alert(document.cookie);x=![];
2 document.vulnerable=x;alert(x);
```

Finally, developers can use an untrusted user data value - contained in a variable `untrustedVar` - in unquoted hyperlink attribute values.

In the following code, the variable `untrustedVar` is inserted inside the unquoted value of the HTML `href` attribute:

```
1 bodyVar.appendChild("<a href=" + untrustedVar +
2 ">link</a>");
3 return <html> ${headVar} ${bodyVar} </html>;
```

The following is an exploit - that mixed percent-encoding and HTML entity encoding - for the above vulnerable code:

```
1 javascript:%26%2397%26%23108%26%23101%26%23114%26
2 %23116%26%2340%26%2334%26%23120%26%23115%26%23115
3 %26%2334%26%2341;
4 //Decoded version
5 javascript:alert("xss")
```

4 Hashed-AST Technique (PLP)

Once that the dataset is built, it can be used to train a deep learning model. However, since the dataset is composed by source code files, a representation strategy is needed. To achieve this goal, we followed the AST-based approach presented by Code2Vec (Alon et al., 2019). Specifically, this representation transverse the AST of a piece of code (i.e. file) in order to obtain all the possible paths between leafs. In this way, the path is represented as a triplet $\langle x_s, p, x_t \rangle$ where x_s is the starting leaf, x_t is the target leaf, and p is the path between them. Each triplet is then mapped to its embedding. Each source code file is represented with the set of embeddings obtained after traversing its AST, and it will be the input for the deep learning algorithm. For a complete description of the representation technique, please refers to (Alon et al., 2019). Since the Code2Vec implementation does not support Hop.js, we extend it by implementing an AST analyzer that obtains the triplets for a given source code file⁴.

Since the number of path between leafs can be very large, we use two parameters to keep the number of triplets into a computationally affordable number:

- `maxPath length/width`: this parameter restrict the obtained paths by the number of nodes between the leaves (length) and the number of branches between the leaves (width).
- `maxContext`: limits the maximum number of triplets used to represent a piece of code.

Regarding the deep learning model (Figure 5), we also use the one used in Code2Vec but changing its output layer to a sigmoid function. In short, the triplets are input into an embedding layer whose output goes into a fully-connected layer. Also, an attention layer is used to learn which paths between leafs

⁴The source code will be available at <https://gitlab.inria.fr/deep-learning-applied-on-web-and-iot-security>

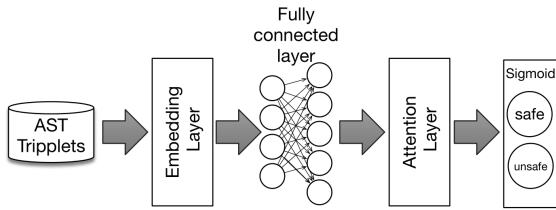


Figure 5: Code2Vec deep learning network used

are more important to detect if a piece of code is affected by XSS. At the end, an output sigmoid layer is used to predict if the piece of code is safe or unsafe. A detailed description of the model can be found at (Alon et al., 2019).

5 Evaluation

Our evaluation process aims to compare the impact of multi-tier languages on Hashed-AST models of supervised deep learning - to detect XSS - with single-tier language. We use Hop.js as a multi-tier language, and Node.js as a single-tier language.

As explained in Section 3.1, to create the dataset we implement a source code generator for Hop.js whereas we modify a Node.js generator implemented by a previous paper (Maurel et al., 2021) to translate into Node.js the Hop.js generated samples.

In this section, we explain the preprocessing step of the database and the experimental protocol (Section 5.1), then, we present and discuss the evaluation results (Section 5.2).

5.1 Evaluation Process

This section explains the preprocessing steps required by the Hop.js and Node.js databases to be “understandable” for the deep learning model.

As shown in Table 1, we split the databases into training (70% of the samples), validation (15%) and testing (15%). To prevent any possible similarities between the generated Hop.js samples, we randomly rename all variables and function names of the datasets.

As explained in Section 4, the Hashed-AST based representation that we employ has two hyperparameter: `maxPath` length/width, and `maxContext`. For `maxPath` we experiment with several values, namely, 10, 20, 30, 50, 80, 130, 210 and 550. Similarly, for `maxContext`, we used 100, 200, 300, 500, 800, 1300, 2100, 3400 and 5500.

By combining the eight `maxPath` values with the nine values of `maxContext`, we train 72 models on the training set for Hop.js and 72 models for Node.js.

We evaluate the 144 models trained with the

validation-set by obtaining the confusion matrix values (FP, FN, TP, TN) to compute the related metrics accuracy, precision, recall, and f-measure. Then, we re-validate these results by using the test-set.

It is important for a vulnerability detector tool not to miss any vulnerability. In this sense, we choose the model that has the highest recall. However, a detector model with perfect recall (i.e. close to 1) but with poor precision (e.g. less than 0.5) means the detector cannot discern if a sample is truly secure and will trigger many false alarms for more than 50% of the secure samples.

In this sense, to analyze the impact of including client-side content as code or text on the Hashed-AST learning phase, we focus our analysis on the evolution of the recall, precision, and f-measure.

5.2 Evaluation Results

In this section, we present the results of our experiment. Due to space constraints, we cannot present all the results obtained. The complete results are available online at https://www.sendgb.com/upload/?utm_source=EFOMAhJfGZb.

We analyze the precision, recall, and f-measure values obtained during the validation and training phases for both Hop.js and Node.js.

Precision distributions analysis Figure 6a shows the precision distributions of Hop.js and Node.js obtained by the 144 models evaluated with the validation and the testing dataset.

For Hop.js, the precision results between the evaluation and the testing phase are very similar. The medians for these distributions are near 72%. Moreover, 25% of the models trained have a high precision between 99% and 86%.

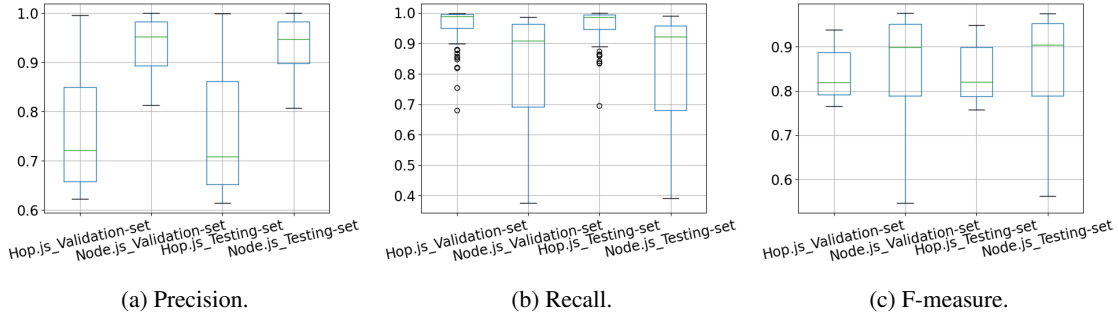
Concerning Node.js precision, the evaluation and testing phases’ results are also similar. The medians for these distributions are near 95% and, some models achieves 100% precision. In fact, 25% of the models trained have a high precision that is between 98% and 100%.

Recall distributions analysis Concerning the Hop.js recall distributions (Figure 6b), the results of the validation and the testing phase are very close. The medians for these distributions are near the maximum recall achieved: 99.80% for the validation set and 99.90% for the testing set.

Concerning the Node.js recall, the evaluation and testing phases’ results are also similar. The medians for these distributions are near 91% and, their maximum recall achieves 99%. Moreover, 25% of the

Table 1: Generated Databases

Language	Database	Classification			Distribution			
		Total	#secure	#insecure	Set	#rule	#secure	#insecure
Hop.js	D1	34,400	18,140	16,260	train	0,1,2,3,4,5	12,998	11,082
	HTML				2,804		2,356	
	validation				2,338		2,822	
Node.js	D1	25800	13,968	11,832	train	0,1,2,3,4,5	9,708	8,352
	HTML				2,144		1,726	
	validation				2,116		1,754	



(a) Precision.

(b) Recall.

(c) F-measure.

Figure 6: Hop.js and Node.js boxplots for the validation set and the testing set distributions.

models trained have a high recall (around 96% and 99%).

F-measure distributions analysis Regarding f-measure results for Hop.js (Figure 6c), the validation and the testing dataset are very similar. The medians for these distributions are near 84%, and, their maximum f-measure achieved is around 94%. In fact, 25% of the models trained have a high f-measure with values between 89% and 94%.

Concerning the Node.js f-measure distributions, the evaluation and testing phases' results are also similar. The medians for these distributions are near 90%, and their maximum f-measure achieved are around 97.50%. The upper 25% of the models have a f-measure ranging 96% and 98%.

To claim any statistically significant difference of these results, a statistical test is needed. We employ the Wilcoxon rank-sum non-parametric test with a probability of error of $\alpha = 0.05$.

We start by analyzing if there is any significant statistically difference between the results of the evaluation and testing phases. For Hop.js we obtained $p_{value-precision} = 0.96$, $p_{value-recall} = 0.70$, and $p_{value-fmeasure} = 0.77$, which means that the Hop.js results, for each metric, in the validation and testing phase are statically similar. We obtain the same conclusion for Node.js with a $p_{value-precision} = 0.86$, $p_{value-recall} = 0.19$ and $p_{value-fmeasure} = 0.15$.

Now we analyze if there is any significant difference between the results of Hop.js and Node.js. For the validation phase results, after applying the tests we obtained a $p_{value-precision} = 7.26E^{-11}$. Thus, there is a significant difference between the Hop.js and the Node.js precision distributions. Meaning that the precision obtained for Node.js is significantly higher. For the recall, we obtained a $p_{value-recall} = 5.57E^{-08}$. Thus, Hop.js has a statistical significant better recall than Node.js. Finally, for f-measure, we conclude that there is no significant difference between the Hop.js and the Node.js after obtaining a $p_{value-fmeasure} = 0.15$.

We also run the same tests for the test-phase results and we reach the same conclusions after obtaining the p-values: $p_{value-precision} = 7.26E^{-11}$, $p_{value-recall} = 3.05E^{-08}$ and $p_{value-fmeasure} = 0.15$.

Finally, we select and compare the best models of the testing phase for each language. As previously explained, each of the metrics analyzed measure some strength of the model. For this reason, we choose the best model for each metric. That is to say, we selected three models for Hop.js and 3 for Node.js (Table 2). While all the models show a good performance, it can be noticed that the Node.js models have the best results.

In summary, taking into account all the results obtained, we found that a better precision was obtained with the Node.js models while a better recall was obtained with the Hop.js models.

Table 2: Comparison of the best Hop.js and Node.js models for each metric

Language	Selected metric	Configuration model		Evaluation phase \%			Testing phase \%		
		maxPath	maxContext	Precision	Recall	F-measure	Precision	Recall	F-measure
Hop.js	precision	30	5500	99.58	82.04	89.97	98.39	86.28	94.94
	recall	50	1300	78.84	99.80	88.09	78.90	99.17	87.89
	f-measure	20	2100	92.40	95.24	93.80	96.75	92.60	94.63
Node.js	precision	30	2100	100	83.75	91.16	95.93	95.58	95.58
	recall	550	5500	95.69	98.69	97.17	96.38	99.05	96.71
	f-measure	80	5500	98.21	96.98	97.59	99.64	95.42	97.48

Along this line, we can conclude that using a multitier language as Hop.js increase drastically the recall despite the lower precision. However, the use of Hop.js does not significantly impact f-measure to claim that using a multitier language positively impacts the XSS identification using deep learning.

Limitations: Although the results are promising, the approach has some limitations. First, this study only focus on applications contained in a single file while most real world applications are divided into several files. Second, the length of the applications analyzed is small. The preprocessing of source code to deep learning approaches using AST is limited by the data size to be analyzed. The larger the data, the more tedious it becomes to perform this preprocessing step. In this type of representation, vector sizes are directly correlated to the size of the source code analyzed in training.

6 Related Work

New deep learning applications on speech recognition and natural languages have motivated recent research in software engineering and cybersecurity communities to apply deep learning to understand vulnerable code patterns and semantics, characterising vulnerable codes. Lin et al. (Lin et al., 2020) review recent literature adopting deep learning approaches to detect software vulnerabilities and identify challenges in this new area.

Maurel et al. (Maurel et al., 2021) compare two different code representations based on Natural Language Processing (NLP) and Programming Language Processing (PLP) for XSS analysis detection in PHP and Node.js. Their deep learning models overcame existing static analyser tools. Our work uses the same Node.js generator for detecting XSS vulnerability with PLP techniques. Different from us, that work did not analyse multitier languages.

Mitch (Calzavara et al., 2019) is a prototype that uses machine learning to black-box detection of CSRF vulnerability. It tries to identify sensi-

tive HTTP requests that require protection against CSRF by manually labelling HTTP requests sent from web applications as sensitive or insensitive HTTP requests.

Neutaint (She et al., 2020) uses AFL fuzzer on programs to generate a list of couples of sources and sinks. Instead of representing statically source code in a vector, Neutaint tries to predict the corresponding taint sinks with a neural network for the specified program. Compared to dynamic taint analysis, the tracked information flow is not obtained from the program’s execution but the neural network.

VulDeePecker (Li et al., 2018b) uses BLSTM neural networks to detect buffer error (i.e., CWE-119) and resource management errors (i.e., CWE-399) related to library/API function calls on C and C++ source code. VulDeePecker used two datasets maintained by the NIST and the SARD project related to buffer and resource management errors in C and C++.

Similarly to VulDeePecker, SySeVR (Li et al., 2018a) uses deep learning to detect vulnerabilities in C/C++ intra-procedural source code using program slicing and Word2vec. As datasets, they used the Software Assurance Reference Data set (SARD) project.

DeepXSS (Fang et al., 2018) proposes an XSS payload detection model based on long-short term memory (LSTM) recurrent neural networks. COD-DLE (Abaimov and Bianchi, 2019) is a deep learning-based intrusion detection prototype to malicious payload related to SQLI and XSS. DeepXSS and COD-DLE learn the difference between a potentially malicious input, which a malicious user can inject into user-controllable input of a web application, from a legitimate input from an ordinary user. Therefore, this type of detector can be used to validate whether user input is vulnerable to XSS or secure before the web application uses it in its program. Unlike our work, the detectors, which we trained, analyze source code that uses input controllable by web application users. They can predict whether a web application is vulnerable to XSS or secure.

Melicher et al. (Melicher et al., 2021) investigate whether machine learning to detect DOM XSS vulnerabilities. They combine Machine Learning and

Taint tracking analyse to reduce the cost of stand-alone taint tracking.

MLPXSS (Mokbal et al., 2019) proposes a neural network-based multilayer perceptron (MLP) to detect XSS attacks. This prototype uses a list of malicious websites and benign websites to generate a raw database. From this database are extracted URL, Javascript, and HTML features, Differently from our work, MLPXSS and Melicher et al. (Melicher et al., 2021) are focused only on client-side code. Moreover, in MLPXSS, the contexts that link the Javascript code, the URLs, and HTML are lost by extracting features independent of each other.

Zhang et al. (Zhang et al., 2020) propose a Monte Carlo Tree Search (MCTS) adversarial example generation algorithm for XSS payloads. MCTS algorithm can only generate adversarial examples of XSS traffic for bypassing XSS payloads detection model. While we analyze the source code to predict if they are vulnerable to XSS, Zhang et al.'s work generates XSS payloads for web traffic.

Shar and Tan (Shar and Tan, 2013) propose an approach to predict whether specific program statements are potentially vulnerable to SQLI or XSS. They developed a prototype tool called PhpMiner1, based on Pixy, for handcrafting 21 features of specific PHP sanitisations of input code. Differently from us, the granularity of this detector is at the instruction level and, the functionality to vectorise the samples has been done manually. Moreover, it is specifically for PHP.

7 Conclusion

In this work, we explore the differences in the XSS detection learning process of Hashed-AST based techniques by using single-tier and multi-tier languages, Node.js and Hop.js. We generated 144 models in one database including HTML/Javascript and CSS as code in Hop.js and 144 models in a database that includes HTML/Javascript and CSS as text. Hop.js obtained a better recall than Node.js despite the lower precision. This implies that our experiments have not shown a major impact on XSS detectors based on deep learning using multitier ASTs compared to ASTs for Node.js.

Our results are promising since they are better than popular static analyzers for JavaScript XSS as shown in previous works (Maurel et al., 2021; AppScan,). For now, our results are based on synthetic databases and we leave as future work the creation of a database to detect XSS in real-world web applications.

Acknowledgment: This research has been partially supported by the ANR17-CE25-0014-01 CISC project, the Inria Challenge SPAI, and CONICET (Argentina) under PIP 2021-2023 id 11220200100430CO. We thank anonymous reviewers for their work and Manuel Serrano for his help with Hop.js.

REFERENCES

- Abaimov, S. and Bianchi, G. (2019). Coddle: Code-injection detection with deep learning. *IEEE Access*, 7:128617–128627.
- Alon, U., Zilberstein, M., Levy, O., and Yahav, E. (2019). code2vec: Learning distributed representations of code. *Proceedings of the ACM on Programming Languages*, 3(POPL).
- AppScan. Appscan scanner for node.js (static mode).
- Balzarotti, D., Cova, M., Felmetzger, V., Jovanovic, N., Kirida, E., Kruegel, C., and Vigna, G. Saner: Composing static and dynamic analysis to validate sanitization in web applications. In *2008 IEEE Symposium on Security and Privacy (sp 2008)*. IEEE. ISSN: 1081-6011.
- Calzavara, S., Conti, M., Focardi, R., Rabitti, A., and Tolomei, G. (2019). Mitch: A machine learning approach to the black-box detection of csrf vulnerabilities. In *IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE.
- Chen, X., Li, M., Jiang, Y., and Sun, Y. (2019). A comparison of machine learning algorithms for detecting XSS attacks. In Sun, X., Pan, Z., and Bertino, E., editors, *Artificial Intelligence and Security - 5th International Conference, ICAIS*, volume 11635 of *Lecture Notes in Computer Science*, pages 214–224. Springer.
- Choi, M., Jeong, S., Oh, H., and Choo, J. (2017). End-to-end prediction of buffer overruns from raw source code via neural memory networks. *CoRR*, abs/1703.02458.
- Cooper, E., Lindley, S., Wadler, P., and Yallop, J. (2006). Links: Web programming without tiers. In *International Symposium on Formal Methods for Components and Objects*, pages 266–296. Springer.
- Dam, H. K., Tran, T., Pham, T., Ng, S. W., Grundy, J., and Ghose, A. (2017). Automatic feature learning for vulnerability prediction. *CoRR*, abs/1708.02368.
- Doupé, A., Cova, M., and Vigna, G. (2010). Why johnny can't pentest: An analysis of black-box web vulnerability scanners. In Kreibich, C. and Jahnke, M., editors, *Detection of Intrusions and Malware, and Vulnerability Assessment, 7th International Conference, DIMVA Proceedings*, volume 6201 of *Lecture Notes in Computer Science*. Springer.
- Fang, Y., Li, Y., Liu, L., and Huang, C. (2018). Deepxss: Cross site scripting detection based on deep learning. In *Proceedings of the 2018 Int. Conf. on Computing and Artificial Intelligence*, pages 47–51. ACM.
- Gundy, M. V. and Chen, H. (2009). Noncespaces: Using randomization to enforce information flow tracking

- and thwart cross-site scripting attacks. In *Proceedings of the Network and Distributed System Security Symposium, NDSS*. The Internet Society.
- Lekies, S., Kotowicz, K., Groß, S., Nava, E. A. V., and Johns, M. (2017). Code-reuse attacks for the web: Breaking cross-site scripting mitigations via script gadgets. In Thuraisingham, B. M., Evans, D., Malkin, T., and Xu, D., editors, *ACM SIGSAC Conference on Computer and Communications Security*. ACM.
- Li, Z. and Zhou, Y. (2005). Pr-miner: automatically extracting implicit programming rules and detecting violations in large software code. *ACM SIGSOFT Software Engineering Notes*, 30(5):306–315.
- Li, Z., Zou, D., Xu, S., Jin, H., Zhu, Y., Chen, Z., Wang, S., and Wang, J. (2018a). Sysevr: A framework for using deep learning to detect software vulnerabilities. *CoRR*, abs/1807.06756.
- Li, Z., Zou, D., Xu, S., Ou, X., Jin, H., Wang, S., Deng, Z., and Zhong, Y. (2018b). VulDeePecker: A deep learning-based system for vulnerability detection. In *Network and Distributed System Security Symposium, NDSS*.
- Lin, G., Wen, S., Han, Q.-L., Zhang, J., and Xiang, Y. (2020). Software vulnerability detection using deep neural networks: A survey. *Proceedings of the IEEE*, 108(10):1825–1848.
- Lin, G., Xiao, W., Zhang, J., and Xiang, Y. (2019). Deep learning-based vulnerable function detection: A benchmark. In *International Conference on Information and Communications Security*, pages 219–232. Springer.
- Lin, G., Zhang, J., Luo, W., Pan, L., Xiang, Y., de Vel, O. Y., and Montague, P. (2018). Cross-project transfer representation learning for vulnerable function discovery. *IEEE Trans. Ind. Informatics*, 14(7):3289–3297.
- Livshits, B. and Chong, S. (2013). Towards fully automatic placement of security sanitizers and declassifiers. In Giacobazzi, R. and Cousot, R., editors, *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013*. ACM.
- Luo, Z., Rezk, T., and Serrano, M. (2011). Automated code injection prevention for web applications. In Mödersheim, S. and Palamidessi, C., editors, *Theory of Security and Applications - Joint Workshop*, volume 6993 of *Lecture Notes in Computer Science*. Springer.
- Maurel, H., Vidal, S., and Rezk, T. (2021). Statically identifying XSS using Deep Learning. In *In Proceedings of the 18th International Conference on Security and Cryptography*, , pages 99–110. SECRYPT.
- Melicher, W., Das, A., Sharif, M., Bauer, L., and Jia, L. (2018). Riding out doomsday: Towards detecting and preventing DOM cross-site scripting. In *25th Annual Network and Distributed System Security Symposium, NDSS*. The Internet Society.
- Melicher, W., Fung, C., Bauer, L., and Jia, L. (2021). Towards a lightweight, hybrid approach for detecting DOM XSS vulnerabilities with machine learning. In *WWW '21: The Web Conference 2021, Virtual Event*, pages 2684–2695. ACM / IW3C2.
- Mokbal, F. M. M., Dan, W., Imran, A., Jiuchuan, L., Akhtar, F., and Xiaoxi, W. (2019). Mlpxss: An integrated xss-based attack detection scheme in web applications using multilayer perceptron technique. *IEEE Access*, 7:100567–100580.
- Node.js (2021). nodejs.org github repository. <https://github.com/nodejs/nodejs.org>.
- OWASP (2021). Cross site scripting prevention cheat sheet. https://cheatsheetseries.owasp.org/cheatsheets/Cross_Site_Scripting_Prevention_Cheat_Sheet.html.
- Russell, R. L., Kim, L. Y., Hamilton, L. H., Lazovich, T., Harer, J. A., Ozdemir, O., Ellingwood, P. M., and McConley, M. W. (2018). Automated vulnerability detection in source code using deep representation learning. *CoRR*, abs/1807.04320.
- Schoepe, D., Balliu, M., Pierce, B. C., and Sabelfeld, A. (2016). Explicit secrecy: A policy for taint tracking. In *IEEE European Symposium on Security and Privacy, EuroS&P*. IEEE.
- Serrano, M. (2006). Hop, multitier web programming.
- Serrano, M., Galesio, E., and Loitsch, F. (2006). Hop: a language for programming the web 2. 0. In *OOPSLA Companion*, pages 975–985.
- Serrano, M. and Prunet, V. (2016). A glimpse of hopjs. In *21th Sigplan Int'l Conference on Functional Programming (ICFP)*, pp. 188–200. ICFP.
- Sestili, C. D., Snively, W. S., and VanHoudnos, N. M. (2018). Towards security defect prediction with AI. *CoRR*, abs/1808.09897.
- Shar, L. K. and Tan, H. B. K. (2013). Predicting SQL injection and cross site scripting vulnerabilities through mining input sanitization patterns. *Inf. Softw. Technol.*, 55(10):1767–1780.
- She, D., Chen, Y., Shah, A., Ray, B., and Jana, S. (2020). Neutaint: Efficient dynamic taint analysis with neural networks. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE.
- Somé, D. F., Bielova, N., and Rezk, T. (2016). On the content security policy violations due to the same-origin policy. *CoRR*, abs/1611.02875.
- Staicu, C.-A., Pradel, M., and Livshits, B. (2018). SYN-ODE: Understanding and automatically preventing injection attacks on NODE.JS. In *Network and Distributed System Security Symposium, NDSS*.
- Wang, S., Liu, T., and Tan, L. (2016). Automatically learning semantic features for defect prediction. In *IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pages 297–308. IEEE.
- Weisenburger, P., Wirth, J., and Salvaneschi, G. (2020). A survey of multitier programming. *ACM Comput. Surv.*, 53(4):81:1–81:35.
- Zhang, X., Zhou, Y., Pei, S., Zhuge, J., and Chen, J. (2020). Adversarial examples detection for XSS attacks based on generative adversarial networks. *IEEE Access*, 8:10989–10996.