HIJaX: Human Intent JavaScript XSS Generator

Yaw Frempong, Yates Snyder, Erfan Al-Hossami, Meera Sridhar and Samira Shaikh
University of North Carolina at Charlotte, 9201 University City Blvd., Charlotte, North Carolina, U.S.A.

Keywords: JavaScript, Cross-site Scripting, Exploit Generation, Natural Language Processing.

Abstract: Websites remain popular targets for web-based attacks such as Cross-Site Scripting (XSS). As a remedy, new research is needed to preemptively secure applications with the use of Automated Exploit Generation (AEG), whereby probing and patching of system vulnerabilities occurs autonomously. In this paper, we present HIJaX, a novel Natural Language-to-JavaScript generator prototype, that creates workable XSS exploit code from English sentences using neural machine translation. We train and test the HIJaX model with a variety of datasets containing benign and malicious intents along with differing numbers of baseline code entries to demonstrate how to best create datasets for XSS code generation. We also examine part-of-speech tagging algorithms and automated dataset expansion scripts to aid the dataset creation and code generation processes. Finally, we demonstrate the feasibility of deploying auto-generated XSS attacks against real-world websites.

1 INTRODUCTION

Cross-Site Scripting (XSS), a OWASP top-ten web attack (owasp (2017)), was the most prominent attack vector of hackers in 2019, comprising nearly 40% of attacks globally (Ilic (2019)). Over 72% of attacks targeted websites, indicating a strong need for continued improvements to web-based security solutions. To combat the persistence and scale of web attacks such as XSS, new defense methods are needed to preemptively secure websites against malicious attacks.

Automatic exploit generation (AEG), an offensive security technique, is a developing field that aims to *automate* the exploit generation process, to explore and test critical vulnerabilities before they are discovered by attackers (Avgerinos et al. (2014)). AEG is also critical for building exploit testbeds for testing defense tools.

In this work, we combine AEG with *Natural Language Processing* (NLP), to enable non-cybersecurity practitioners to detect vulnerabilities in their own software—by issuing commands, and with an interface translating and executing command operations (Dheap (2017)). To this end, we report on the prototype HIJaX, a *Human Intent JavaScript XSS generator*, that adapts neural machine translation to translate natural language sentences bearing malicious intentions (*intents*) into exploit code. An *intent* describes an action in natural language that a user wants to implement, such as `visit website X` or `get all strings from array Y`. A *snippet* represents the code required to perform the action specified in the intent.We focus on using NLP to generate XSS attack code (snippets) based on natural language intents. The main contributions of our work include:

- We are the first to combine NLP and AEG with JavaScript to generate XSS code.
- We design and build eleven different datasets for training the HIJaX model; we use two approaches for this—*manual selection* and *transpiling*—with differing degrees of variance.
- We demonstrate that HIJaX can generate code with high accuracy (using BLEU, exact, syntax, execution scores).
- We provide an automated means to expand a dataset while maintaining the original qualities of the *baseline entries* (the original intent-snippet pairs mined from online sources and used to create our datasets (see Section §2.1)).
- We design a *part-of-speech tagging* (POS tagging) algorithm for website names and URLs, which resulted in a significant increase in code generation accuracy for small datasets containing malicious code.
- Our JavaScript Syntax & Execution Tester validates the syntax and execution of JavaScript code.
- Our XSS Attack Tester validates the success of XSS attacks by deploying XSS attack code

Frempong, Y., Snyder, Y., Al-Hossami, E., Sridhar, M. and Shaikh, S.
HIJaX: Human Intent JavaScript XSS Generator.
DOI: 10.5220/0010583807980805
In *Proceedings of the 18th International Conference on Security and Cryptography (SECRYPT 2021)*, pages 798-805
ISBN: 978-989-758-524-1

generated by HIJaX on the following vulnerable websites: WebGoat (Owasp (2020)), BWAPP (Mesellem (2018)), & The 12 Exploits of XSS-mas (Chef Secure (2019)).

Motivations. NLP-based AEG is a nascent field that has rich potential to provide unique and effective opportunities for automated exploit construction. Some unique NLP-based AEG features include:

- *Textual Information for Attack Construction:* Source code analysis is often insufficient to generate exploits practically. To generate difficult exploits AEG systems tend to reason about binary and runtime details which are often not captured in source code. NLP-based AEG approaches such as SemFuzz (You et al. (2017)) propose novel algorithms that capture such details from textual sources such as CVE and git log reports to generate difficult exploits that are not easy to detect and patch on the source code level.
- *Exploit Abstraction:* Utilizing NLP-based AEG enables further abstraction of exploits in AI models. Often a single natural language description can map onto multiple exploit source codes. This enables NLP models to generate more exploits from a single dataset and also facilitates further research on abstracting exploits.
- *Interpretability:* NLP-based AEG pairs abstract and interpretable descriptions with source code to generate cryptic exploits. This can help build a bridge to further human understanding of machine exploits. This can assist experts in creating abstract ontologies for exploits. We also hope that our work can be applied so that non-cyber security experts can interpret and even generate exploits. This helps keep developers in the loop about security issues during development.

HIJaX can generate a range of XSS attacks that it has been trained on, if given a sufficient description of the attack. HIJaX can help developers early in the software development life cycle by helping generate exploit code to test possible vulnerabilities. In future work, we plan to augment HIJaX with the ability to generate corresponding defense code.

Previous works either explore natural language to code transformation (Ling et al. (2016); Barone and Sennrich (2017); Gordon and Harel (2009)), or use NLP techniques to find and generate exploits (You et al. (2017)), but no extant research focuses on directly mapping Natural Language intents to exploit code. To our knowledge, our work is the first to explore this subsection of NLP and AEG.

The rest of this paper is organized as follows. Section 2 provides an overview of HIJaX, our dataset creation processes, and metrics for evaluation. Section 3 describes experiment results, §4 discusses related work, and §5 concludes.

2 THE HIJaX PROTOTYPE

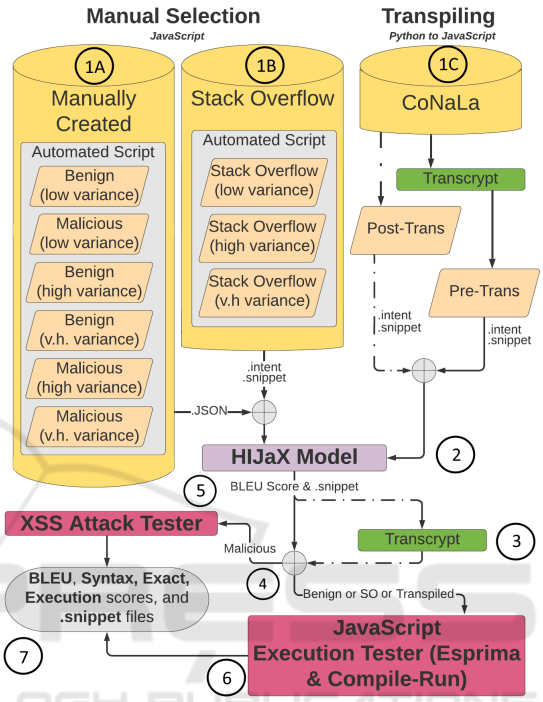

Figure 1: HIJaX Full Code-Generation System.

HIJaX is a prototype NLP-based code generation and validation tool. The code generation component in HIJaX (the HIJaX model (Anonymous (2020))) is built using techniques from neural machine translation, specifically an English-to-Python Translation Model built for the CoNaLa Challenge (Carnegie Mellon University (2019)). The CoNaLa Challenge tests a system's ability to generate Python code when given an English sentence or phrase. The HIJaX model processes English intents, provided by our datasets, into *JavaScript* snippets. The ability to translate an intent into its equivalent snippet allows us to translate malicious intents into exploit code.

HIJaX uses sequence-to-sequence architecture (Ma'amari (2018)) & a bi-directional LSTM as the encoder to transform an embedded intent sequence into a vector of hidden states with equal length. We implement this architecture with Bahdanau-style attention (Bahdanau et al. (2015)) using `xnmt` (Neubig et al. (2018)). We use an Adam optimizer (Kingma and Ba (2014)) with $\beta_1 = 0.9$ & $\beta_2 = 0.999$. We use Auto-Regressive Inference

components with beam search (beam size of 5).

Fig. 1 shows an overview of our tool chain. A key effort in this work is the process of building adequate datasets for training the HIJaX model on JavaScript (exploit) code. We employ two approaches for this effort—*manual selection* (manually collecting training data from online JavaScript sources) and *transpiling* (converting existing Python snippets from the CoNaLa dataset (Carnegie Mellon University (2019)) into JavaScript). The orange boxes in Fig. 1 show datasets; Steps 1A and 1B show manually selected datasets, and Step 1C shows the transpiled dataset. In Step 1A, we create our own JavaScript intent-snippet pairs manually from a JavaScript tutorial website and a GitHub repository of XSS payloads, resulting in the *Benign* and *Malicious* datasets respectively. We then expand the collected datasets with automated scripts to create variations of the chosen snippet code structures. In Step 1B, we scrape JavaScript intent-snippet pairs manually from Stack Overflow (SO) (Overflow" (2020)), a website where developers can ask questions and receive answers to programming problems, then expand the collected dataset with automated scripts. This results in the *Stack Overflow* dataset. In Step 1C, we convert an existing Python dataset (Carnegie Mellon University (2019)) into JavaScript through transpiling. We use two approaches for transpiling: The first approach is *Pre-transpiling*, where we convert the Python dataset into JavaScript before using it as input for HIJaX. The second approach is *Post-transpiling*, where we use the Python dataset as input for HIJaX then converting it to JavaScript later. In Step 2, 80% of each dataset is used to train the HIJaX model and 20% is used to test the HIJaX model. The model outputs the generated snippets along with BLEU and exact scores. In Step 3, the generated snippets from the Post-transpiled datasets are converted to JavaScript. In Step 4, the datasets are tested for syntax and execution using different methods based on the type of dataset they are. The generated snippets from the malicious datasets are used as input for the Attack Tester. Generated snippets from the Stack Overflow, Benign, and Transpiled datasets are used as input for the JavaScript Syntax and Execution Tester. In Step 5, the Attack Tester outputs the syntax and execution score for the XSS attacks in the malicious datasets. In Step 6, the JavaScript Syntax and Execution Tester outputs the syntax and execution score for the JavaScript code in the Benign, Stack Overflow, and Transpiled datasets. In step 7, we compare all the datasets to each other based on their BLEU, exact, syntax, and execution score.

2.1 Manual Selection

We coin the term *manual selection* for our approach of creating datasets to train the HIJaX model, where we manually collect data from online sources such as Stack Overflow (SO). We use SO to collect large numbers of JavaScript-based questions and answers, since it offers a good representation of how a user would describe a piece of code they want to generate. These JavaScript-based questions and answers are useful for training HIJaX, a tool that converts descriptions of code into code, since the questions often contain a precise description of code and the most popular answer to the question often contains the correct corresponding snippet of code. We also manually select snippets from a JavaScript tutorial website (HTMLCheatSheet (2019)) and a list of XSS payloads from a GitHub repository (Payload Box (2019)) to create the baseline entries for the Benign and Malicious datasets (see more details in §2.3).

2.2 Transpiling

We coin the term *transpiling* for our approach of creating datasets to train the HIJaX model, where we convert existing Python snippets to JavaScript. While the manual selection process tests HIJaX with JavaScript datasets, exploring the capability to utilize existing large datasets written in other programming languages and then translating them into JavaScript is beneficial from a cost and time efficiency perspective.

As mentioned earlier, we conduct two experiments, Pre-transpiled and Post-transpiled, with the existing Python dataset (Carnegie Mellon University (2019)) (called `conala-mined`, but we refer to it as CoNaLa) to determine if HIJaX can generate JavaScript code even if the dataset contains snippets in a programming language other than JavaScript.

We use a transpiler called Transcrypt (Transcrypt (2016)) to transform the Python CoNaLa dataset into its JavaScript equivalent. The CoNaLa dataset consists of 598,237 intent/snippet pairs.

In our transpiling process, as depicted in Fig. 1, we first retrieve the `.intent` and `.snippet` files from the CoNaLa dataset. In our post-transpiled experiment, we train the HIJaX model with the files directly, and the model outputs a Python `.snippet` file. We then use Transcrypt to transpile the Python `.snippet` file into its JavaScript equivalent. In the pre-transpiled experiment, we transpile CoNaLa's Python `.snippet` into its JavaScript equivalent then train the model.

```
XSS
-----------------------------------------------------------------------
intent: redirect script to var0
snippet: <script>window.location="https://www.var0"</script>

var0: facebook.com

intent: redirect script to facebook.com
snippet: <script>window.location="https://www.facebook.com"</script>


Stack Overflow
-----------------------------------------------------------------------
intent: Replace all occurences of String var0 with String var1 using REGEX
snippet: anotherString = someString.replace(/var0/g, 'var1');

var0: cat
var1: dog

intent: Replace all occurences of String cat with String dog using REGEX
snippet: anotherString = someString.replace(/cat/g, 'dog');
```

Figure 2: Dataset Expansion.

2.3 Datasets

In addition to the Pre-transpiled and Post-transpiled datasets (see §2.2), we also create nine datasets within the umbrella of manual selection. Three of the manual selection datasets are created with the use of SO while the remaining six are created from other sources (as described in §2.1). The following datasets are included in manual selection since their baseline entries are manually selected or created prior to expanding the dataset with an automated script: *Stack Overflow Low Variance*, *Stack Overflow High Variance*, *Stack Overflow Very-High Variance*, *Benign Low Variance*, *Benign High Variance*, *Benign Very-High Variance*, *Malicious Low Variance*, *Malicious High Variance*, and *Malicious Very-High Variance*.

"Benign" refers to intent-snippet pairings that are representative of simple programmatic operations, whereas "malicious" refers to intent-snippet pairings that are XSS-related, and are generally more abstract or semantically complex. *Variance* is a quantitative measurement referring to the number of baseline entries present in a dataset; whereby Low is one, High is five, and Very-High is fifty. Baseline entries for the Benign and Malicious datasets are characterized in the same way as baseline entries in the SO dataset from the manual selection process, and are used to enlarge a dataset. As depicted in Fig. 2, we increase the size of our datasets by first taking a dataset's baseline entry and changing the identified variables with randomly selected words from a word bank (FreeBSD (2020)) to generate multiple similar snippets for training.

2.4 POS Tagging

POS Tagging, in combination with regular expressions, allows for an automated way to mark website names in an intent—identifying website names is important since they often appear in XSS attacks. POS tagging is an NLP technique for marking words in text and grouping them into a particular part of speech (nouns, verbs, adjectives, adverbs, pronouns, etc). We use the POS tagging function in the `spaCy` Python library ("spaCy" (2020)) to mark pronouns as websites. To mitigate mislabeling, we mark all nouns that succeed a verb in an intent as a website. This is effective for Malicious datasets because they consist of URL redirects that often contain phrases such as "Go to (website)" or "Redirect to (website)".

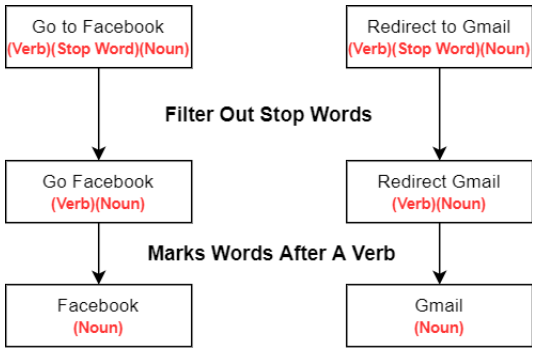

Figure 3: Tagging website names with POS tagging.

2.5 Evaluation of Generated Code

We use three methods to evaluate the snippets that are generated from HIJaX:

BLEU Score: *BLEU* score (Carnegie Mellon University (2019)) is a metric for comparing generated translations of text to reference translations. In our case, the generated translations are the snippets that HIJaX generates and the reference translations are the intent-snippets pairs provided as input to HIJaX. BLEU score is generated by calculating a precision value, based on word sequences, between the provided snippet and the predicted snippet in addition to a *brevity penalty* (Tatman" (2019)).

Exact Score: Unlike BLEU score, where precision and brevity are used to measure generated snippets against provided snippets, the exact score is based simply on the whether or not the generated snippet is an exact match to the provided snippet.

Syntax & Execution Score: In the third method, we use our own *syntax & execution score* to evaluate if generated snippets are syntactically valid and executable using our JavaScript Syntax & Execution Tester and XSS Attack Tester. The purpose of the JavaScript Syntax & Execution Tester is to measure HIJaX's ability to generate JavaScript code that has correct syntax and executes without error. The purpose of the XSS Attack Tester is to measure HIJaX's

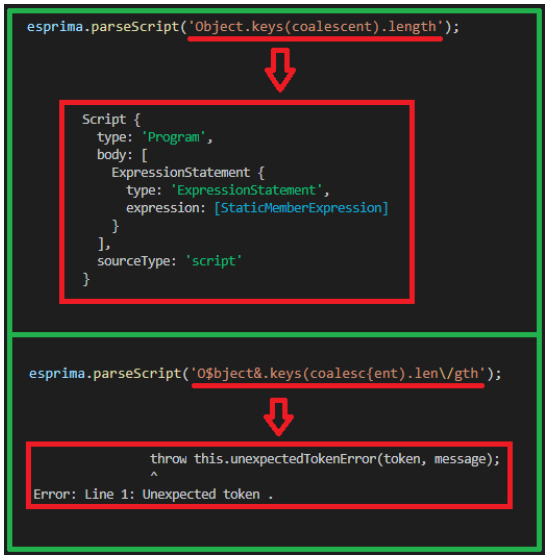

Figure 4: Esprima outputs - success & failure.

ability to generate XSS attacks that execute without error and work as intended.

Our JavaScript Syntax & Execution Tester is built using the Node.JS packages Esprima JavaScript Tokenizer (Ariya Hidayat (2020)) (see Fig. 4) and compile-run (Vibhor Agrawal (2019)). In our JavaScript Syntax & Execution Tester, we take in a list of snippets written in JavaScript and output syntax and execution scores based on the number of snippets that had correct JavaScript syntax and executed without error. We use the Esprima JavaScript Tokenizer to identify variables and functions in a snippet. Tokenization is important because it allows us to determine which identifiers need to be initialized. We initialize every variable and function in a snippet with every combination of default values (zero, empty array, or empty string) so it has the chance to compile and execute without error. Initialization allows us to know that a failed execution is a result of the snippet itself and not uninitialized variables and functions. We then use compile-run to compile our JavaScript snippets. Based on the response from compile-run, we calculate the number of successful snippet executions in the SO and Benign datasets.

In our XSS Attack Tester, we use Selenium (Selenium (2020)), a tool that enables us to automate actions in a web browser, to inject the XSS attacks generated from HIJaX into targeted input fields and validate the success of each XSS attack by detecting whether the correct response is generated from the browser or server. We test the execution of XSS attacks in the Malicious datasets using multiple insecure websites made for penetration testing: WebGoat, BWAPP, and The 12 Exploits of XSS-mas. The Malicious dataset contains XSS attacks that open alerts, prompts, & confirmation boxes in the browser, redirect to other websites, send user data to external servers, and so on. These websites contain unsanitized input text boxes where users can inject attack code to generate some result. We use a socket server to listen for incoming data being sent to an IP address. This is used to see if XSS attacks that send data to a remote server has successfully executed.

3 PRELIMINARY EXPERIMENTAL RESULTS

Setup. As mentioned earlier in §2.1, for all datasets we define Low Variance as a dataset made up of one baseline entry, High Variance as a dataset made up of five baseline entries, and Very-High Variance as a dataset made up of 50 baseline entries. For the Benign, Malicious, and SO datasets, we define *Small* as a dataset made up 200 entries, *Medium* as 2,000 entries, and *Large* as 20,000 entries. For the Transpiled datasets we define *Small* as a dataset made up 1,500 entries, *Medium* as 15,000 entries, and *Large* as 150,000 entries. To remind the reader, Pre-Transpiled is a dataset of JavaScript code that is transpiled from the (Python) CoNaLa Dataset before being fed into the HIJaX model, and Post-Transpiled is a dataset of JavaScript code that is transpiled from the (Python) CoNaLa Dataset after being fed into the HIJaX model (see §2.2).

Training & Testing. We use our eleven different datasets to train our HIJaX models (see Section §2.1). If we want a model to generate exploit code based on intents that describe exploits, we train using one of our malicious datasets which contains 200, 2,000, and 20,000 examples of malicious intent-snippet pairs. In order to measure our model's performance, we split our dataset into two subsets, the testing set and the training set. The training set, which contains 80% of the intent-snippet pairs in the original dataset, is used to train the model. The testing set, which contains the other 20% of the intent-snippet pairs in the original dataset, is used to test the model's performance. We use the testing set to measure our model's performance because we already know the correct snippet for each intent in the testing set. We can use the testing set to see if the model can predict the correct snippet for each intent in the testing set.

Experiments. To see the effect of size, variance, & POS tagging on code generation accuracy, we conduct the following tests: **Experiment #1:** Benign (Variance): Low vs. High vs. Very–High; **Experiment #2:** Benign (Size): Small vs. Medium vs. Large; **Experiment #3:** SO (Variance): Low vs. High vs. Very–High; **Experiment #4:** SO (Size): Small vs. Medium vs. Large; **Experiment #5:** Malicious: Low vs. High vs. Very–High; **Experiment #6:** Malicious: Small vs. Medium vs. Large; **Experiment #7:** Malicious: POS–Tagging Disabled vs. POS–Tagging Enabled; **Experiment #8:** Transpiled (Before & After): Pre–Transpiled vs. Post–Transpiled; **Experiment #9:** Transpiled (Size): Small vs. Medium vs. Large; **Experiment #10:** XSS Execution (Variance): The 12 Exploits of XSS-mas vs. WebGoat vs. BWAPP; **Experiment #11:** XSS Execution (Size): The 12 Exploits of XSS-mas vs. WebGoat vs. BWAPP.

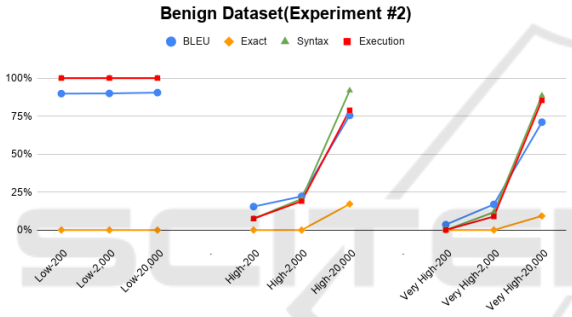

Figure 5: Benign Experiment Results Across Size.

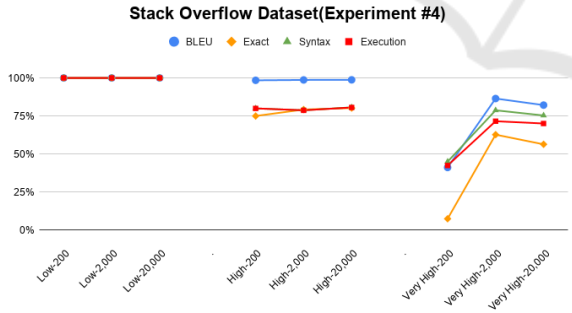

Figure 6: SO Experiment Results Across Size.

Results. In Experiment #1 & #2 (see Fig. 5), increasing variance had an overall negative effect on all Benign datasets but had a less significant effect on datasets of size 20,000. In Experiments #3, #4, #5, #6, & #7 (see Fig. 6 & Fig. 7), increasing variance had an overall negative effect on all datasets and a less significant effect on datasets of size 2,000 and 20,000. In Experiments #10 & #11, we saw similar results for successful XSS execution on WebGoat, BWAPP, & The 12 Exploits of XSS-mas. In Experiment #8 (see

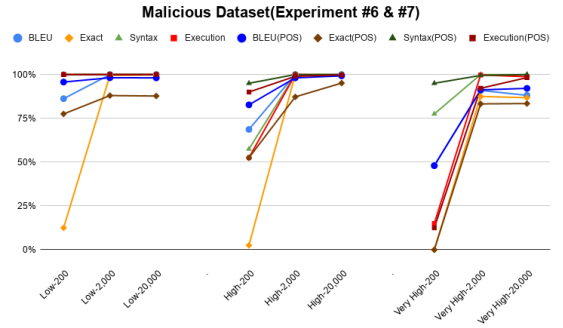

Figure 7: Malicious Experiment Results Across Size.

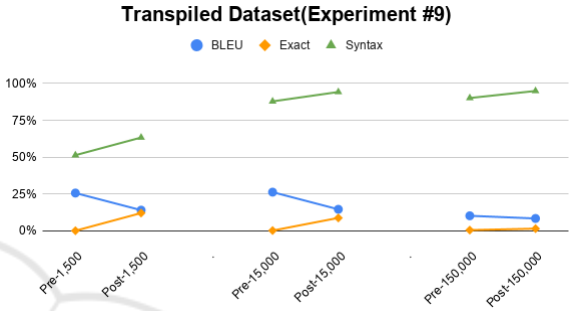

Figure 8: Transpiled Experiment Results Across Size.

Fig. 8), Pre-Transpiled performed roughly twice as well in BLEU score compared to Post-Transpiled. In Experiment #9, exact and syntax score are positively correlated to dataset size. Due to space constraints we are only able to present figures for select experiments.

We saw that increasing variance had an overall negative effect on all datasets but had a less significant effect on datasets of size 2,000 and 20,000. We also saw that POS tagging measurably improved execution scores for smaller malicious datasets. We concluded that a medium dataset is sufficient for generating different executable XSS attacks such as URL redirects, popups, alerts, and cookie stealing attacks. We also concluded that HIJaX cannot accurately generate JavaScript code through the use of transpiling. Given these preliminary results, we expect that with a more robust and diverse malicious dataset of sufficent size, HIJaX would be able to generate a wide-range of executable XSS attacks with high accuracy.

4 RELATED WORK

NLP for Code Generation. Natural language for code generation has been explored with JavaScript, Python (Barone and Sennrich (2017); Ling et al. (2016)), C (Mokhov et al. (2014))], C# (Iyer et al. (2016); Allamanis et al. (2015)), C++ (Mokhov et al.

(2014)), Java (Mokhov et al. (2014); Ling et al. (2016)), and SQL (Iyer et al. (2016); Giordani and Moschitti (2012)). While these works conduct code generation that is a representation of natural language, they do not incorporate aspects of exploit generation, which we do by training our tool on XSS-related data. To our knowledge, there is no other available research on NLP-to-code for exploit generation, in which the natural language intent describes an attack, and its corresponding snippet contains the executable code.

JavaScript & XSS Code Synthesis. To our knowledge, only one work exists for translating English intents into JavaScript code— a natural-language programming system for a computer video game that transforms user sentences into JavaScript code (Hsiao (2018)). Past works utilize XSS attack generation for creating test cases. AppSealer focuses on automatically generating security patches on Android apps when vulnerabilities are detected (Zhang and Yin (2014)). Another work automates unit testing to detect XSS vulnerabilities caused by improper encoding (Mohammadi et al. (2017)). In another work, researchers also build a novel and principled type-qualifier based mechanism that attempts to automate the process of sanitization for XSS attack prevention (Samuel et al. (2011)). The approaches stated above utilize XSS attack generation but do not tie the attack code to intents written by people. With HIJaX, we connect the human intention behind the XSS attack to the attack code using NLP.

5 CONCLUSION & FUTURE WORK

We present HIJaX: a prototype NLP-AEG tool that adapts techniques from machine translation to generate JavaScript, especially XSS code, from written intents. Our contributions include the creation of eleven datasets using two different approaches, automating the testing of output snippets for correct JavaScript syntax & execution, and successful execution of XSS attacks generated by HIJaX. We also contribute an algorithm for POS tagging website names and URLs to increase the code generation accuracy of XSS-based intent-snippet pairs. Our preliminary experimental results show that HIJaX is able to generate both executable benign JavaScript code and malicious JavaScript code that successfully executes in our chosen online testing environments: WebGoat, BWAPP, & The 12 Exploits of XSS-mas. This prototype of HIJaX has laid the groundwork for our aim to use NLP-based AEG to help developers create more secure websites early in the software development life cycle and help security experts automated the process of generating real-life XSS vulnerabilities.

In future work, we plan to construct larger datasets with code examples of the most popular XSS attacks as well as adapt the model to website-specific DOM structure. This will allow us to generalize HIJaX and generate semantically new attacks as long as the intent describes a type of XSS attack that HIJaX is familiar with such as phishing, keylogging, etc. We also plan to have HIJaX not only generate attack code but also defense code that prevents the attack from executing. Developers can use the generated defense snippet to proactively secure their website in the design & prototyping stage of the software development life cycle.

Since our approach of NLP-based AEG is novel, we plan to compare HIJaX's code generation performance against different neural machine translation techniques. This includes LSTM, GRU, and RNN seq2seq techniques (Pedamallu" (2020)).

REFERENCES

Allamanis, M., Tarlow, D., Gordon, A., and Wei, Y. (2015). Bimodal Modelling of Source Code and Natural Language. In *Proc. of the International Conference on Machine Learning (ICML)*, pages 2123–2132.

Anonymous (2020). HIJaX model. https://github.com/HIJaXAnonymousRepo/HIJaX. Retrieved 04-01-2021.

Ariya Hidayat (2020). Esprima: ECMAScript parsing infrastructure for multipurpose analysis. https://esprima.readthedocs.io/en/latest/syntactic-analysis.html. Retrieved 04-01-2021.

Avgerinos, T., Cha, S. K., Rebert, A., Schwartz, E. J., Woo, M., and Brumley, D. (2014). Automatic exploit generation. *Communications of the Association for Computing Machinery (Commun. ACM)*, page 74–84.

Bahdanau, D., Cho, K., and Bengio, Y. (2015). Neural machine translation by jointly learning to align and translate. *Computing Research Repository (CoRR) on Arxiv.org*, pages 1–15.

Barone, A. V. M. and Sennrich, R. (2017). A Parallel Corpus of Python Functions and Documentation Strings for Automated Code Documentation and Code Generation. In *The 8th Intl. Joint Conf. on Natural Language Processing (IJCNLP 2017)*, pages 314–319.

Carnegie Mellon University (2019). CoNaLa: The Code/-Natural Language Challenge. https://conala-corpus.github.io/. Retrieved 04-01-2021.

Chef Secure (2019). The 12 Exploits of XSS-mas. https://playground.insecure.chefsecure.com/the-12-exploits-of-xssmas. Retrieved 04-01-2021.

Dheap, V. (2017). Man With Machine: Harnessing the Potential of Artificial Intelligence. https://securityintelligence.com/

man-with-machine-harnessing-the-potential-of\
-artificial-intelligence. Retrieved 05-13-2021.

FreeBSD (2020). The FreeBSD Project. https://www.freebsd.org/. Retrieved 04-01-2021.

Giordani, A. and Moschitti, A. (2012). Translating Questions to SQL Queries with Generative Parsers Discriminatively Reranked. In *Proc. of Intl. Conf. on Computational Linguistics 2012: Posters - (COLING)*, pages 401–410.

Gordon, M. and Harel, D. (2009). Generating Executable Scenarios from Natural Language. In *Intl. Conf. on Computational Linguistics and Intelligent Text Processing (CICLing)*, pages 456–467.

Hsiao, M. S. (2018). Automated Program Synthesis from Object-Oriented Natural Language for Computer Games. In *Controlled Natural Language - Proc. of the Sixth Intl. Workshop (CNL)*, pages 71–74.

HTMLCheatSheet (2019). JS CheatSheet. https://htmlcheatsheet.com/js/. Retrieved 04-01-2021.

Ilic, J. (2019). Cross-Site Scripting (XSS) Makes Nearly 40% of All Cyber Attacks in 2019. https://www.precisesecurity.com/articles/cross-site-scripting-xss-makes-nearly-40-of\
-all-cyber-attacks-in-2019/. Retrieved 04-01-2021.

Iyer, S., Konstas, I., Cheung, A., and Zettlemoyer, L. (2016). Summarizing Source Code using a Neural Attention Model. In *Proc. of the Assoc. for Computational Linguistics (Volume 1: Long Papers)*, pages 2073–2083.

Kingma, D. P. and Ba, J. (2014). Adam: A Method for Stochastic Optimization. *Intl. Conf. on Learning Representations*, pages 1–15.

Ling, W., Blunsom, P., Grefenstette, E., Hermann, K. M., Kočiský, T., Wang, F., and Senior, A. (2016). Latent Predictor Networks for Code Generation. In *Proc. of the Assoc. for Computational Linguistics (Volume 1: Long Papers) (ACL)*, pages 599–609.

Ma'amari, M. (2018). NLP — Sequence to Sequence Networks — Part 2 — Seq2Seq Model (EncoderDecoder Model). https://towardsdatascience.com/nlp-\sequence-to-sequence-networks-part-2\
-seq2seq-model-encoderdecoder-model-6c22e29fd7e1.

Mesellem, M. (2018). bWAPP - an extremely buggy web app! http://www.itsecgames.com/. Retrieved 04-01-2021.

Mohammadi, M., Chu, B., and Lipford, H. R. (2017). Detecting Cross-Site Scripting Vulnerabilities through Automated Unit Testing. In *2017 Intl. Conf. on Software Quality, Reliability and Security (QRS)*, pages 364–373.

Mokhov, S. A., Paquet, J., and Debbabi, M. (2014). The Use of NLP Techniques in Static Code Analysis to Detect Weaknesses and Vulnerabilities. In *Canadian Conf. on Artificial Intelligence (CAIAC)*, pages 326–332.

Neubig, G., Sperber, M., Wang, X., Felix, M., Matthews, A., Padmanabhan, S., Qi, Y., Sachan, D., Arthur, P., Godard, P., Hewitt, J., Riad, R., and Wang, L. (2018). XNMT: The eXtensible Neural Machine Translation Toolkit. In *Proc. of the 13th Conference of the Association for Machine Translation in the Americas (Volume 1: Research Track) (AMTA)*, pages 185–192.

Overflow", S. (2020). "stack Overflow: Where Developers Learn Share & Build Careers". https://stackoverflow.com/. Retrieved 04-01-2021.

owasp (2017). OWASP Top Ten. https://owasp.org/www-project-top-ten/. Retrieved 04-01-2021.

Owasp (2020). OWASP WebGoat - Learn the hack - Stop the attack. https://owasp.org/www-project-webgoat/. Retrieved 04-01-2021.

Payload Box (2019). Cross Site Scripting (XSS) Vulnerability Payload List. https://github.com/payloadbox/xss-payload-list. Retrieved 04-01-2021.

Pedamallu", H. (2020). "RNN vs GRU vs LSTM". https://medium.com/analytics-vidhya/rnn-vs-gru-vs-lstm-863b0b7b1573.

Samuel, M., Saxena, P., and Song, D. (2011). Context-Sensitive Auto-Sanitization in Web Templating Languages Using Type Qualifiers. In *Proc. of the Association for Computing Machinery Conf. on Computer and Communications Security (ACM CCS)*, page 587–600.

Selenium (2020). SeleniumHQ Browser Automation. https://www.selenium.dev/. Retrieved 04-01-2021.

"spaCy" (2020). "spaCy · Industrial-strength Natural Language Processing in Python". https://spacy.io/.

Tatman", R. (2019). "Evaluating Text Output in NLP: BLEU at your own risk". https://towardsdatascience.com/evaluating-text-output-in-nlp-bleu-at-your-\
own-risk-e8609665a213.

Transcrypt (2016). https://www.transcrypt.org/docs/html/index.html. Retrieved 04-01-2021.

Vibhor Agrawal (2019). compile-run. https://www.npmjs.com/package/compile-run. Retrieved 04-01-2021.

You, W., Zong, P., Chen, K., Wang, X., Liao, X., Bian, P., and Liang, B. (2017). SemFuzz: Semantics-Based Automatic Generation of Proof-of-Concept Exploits. In *Proc. of the Conf. on Comp. and Comm. Security (ACM CCS)*, page 2139–2154.

Zhang, M. and Yin, H. (2014). AppSealer: Automatic Generation of Vulnerability-Specific Patches for Preventing Component Hijacking Attacks in Android Applications. In *Network and Distributed System Security Symposium 2014 (NDSS)*.