LL-XSS: End-to-End Generative Model-based XSS Payload Creation

LL-XSS: End-to-End Generative Model-based XSS Payload Creation

Sohail Khan

Computer Science Department, Effat College of Engineering
Effat University
Jeddah, KSA
sohkhan@effatuniversity.edu.sa

***Abstract*—In the realm of web security, there is a growing shift towards harnessing machine learning techniques for Cross-Site Scripting (XSS) vulnerability detection. This shift recognizes the potential of automation to streamline identification processes and reduce reliance on manual human analysis. An alternative approach involves security professionals actively executing XSS attacks to precisely pinpoint vulnerable areas within web applications, facilitating targeted remediation. Furthermore, there has been a growing interest in machine learning-based methods for creating XSS payloads in academic and research domains. In this research, we introduce a new model for generating XSS payloads, utilizing a combination of auto-regressive and generative AI models to craft malicious scripts intended to exploit potential vulnerabilities. Our approach to XSS vulnerability detection encompasses both frontend and backend code, providing organizations with a comprehensive means to enhance web application security.**

I. Introduction

Cross-Site Scripting (XSS) attacks fall under the category of injection attacks. In these attacks, malicious scripts are introduced into websites that users trust. They occur when an attacker takes advantage of vulnerabilities in a web application by inserting harmful code, typically in the form of a script that runs in a web browser, and then shares it with other users. The success of XSS attacks is often due to weaknesses in web applications that do not properly check or secure user inputs before displaying them to others [1].

The injected malicious script deceives a user's web browser into thinking it comes from a legitimate source, causing the script to run. Consequently, the attacker gains unauthorized access to sensitive information stored in the user's browser, such as cookies or session tokens. Moreover, the malicious script can alter the content of the web page, potentially leading to serious consequences.

Traditional methods for identifying Cross-Site Scripting (XSS) vulnerabilities have relied on manual analysis conducted by human experts [1], [2]. This approach necessitates scrutinizing both the backend and frontend code of web applications, a process that is not only time-consuming but also comes with substantial financial costs. The need for human intervention has been a significant bottleneck in addressing XSS vulnerabilities effectively [3].

In recent times, there has been a shift towards utilizing machine learning techniques to detect XSS vulnerabilities [4]–[6]. This shift acknowledges the potential of automation to streamline the identification process and reduce the reliance on human expertise. However, an alternative perspective on addressing the XSS problem has emerged: the idea of conducting red-teaming attacks to actively attempt XSS exploitation. By attempting to execute XSS attacks, security professionals can pinpoint the exact areas within web applications that are susceptible to such attacks.

Machine learning-based XSS payload generation has gained attention and popularity in the academic and research literature [7], [8]. This method focuses on automating the creation of malicious scripts that can trigger XSS vulnerabilities. In the context of this research paper, we propose a novel model for XSS payload generation that leverages a combination of auto-regressive models and generative AI models. Auto-regressive models, known for their robustness and versatility, have played a pivotal role in the development of large language models like ChatGPT and Bard. Our approach aims to harness the power of these models to enhance XSS detection and mitigation efforts in web application security.

Our proposed model offers a comprehensive approach to XSS vulnerability detection within an organization's web-based system. It begins by taking into account both the frontend and backend code of the system. These components are subjected to rigorous analysis using auto-regressive models, which excel at processing and understanding complex code structures. Subsequently, the results of these individual analyses are combined, providing a holistic view of the web application's security posture.

This combined analysis then serves as the foundation for automating the creation of XSS payloads through the utilization of generative AI techniques. By leveraging generative AI, our model generates malicious scripts designed to exploit potential vulnerabilities identified during the analysis phase. This automated process significantly accelerates the identification of XSS vulnerabilities compared to traditional manual inspection methods.

Paper Organization: Rest of the paper is organized as follows:

979-8-3503-9356-9/24/$31.00 ©2024 IEEE

Section II presents required background information related to cross-site scripting and large language models, Section III details the proposed architecture – LL-XSS. Results and discussion are presented in Section IV, Section V examines related work within the field, and Section VI concludes the paper.

II. Background

A. Cross-site Scripting

1) Principles of Cross-Site Scripting: Cross-Site Scripting (XSS) vulnerabilities represent a critical security issue within web applications. Essentially, XSS is a type of injection attack where malicious actors inject code, typically in the form of scripts, into web pages viewed by other users. This code can execute within the context of the victim's browser, potentially leading to various malicious actions, such as data theft, session hijacking, or defacement of the website [1], [2].

These vulnerabilities are primarily caused by developers failing to implement effective input validation and output encoding mechanisms at multiple levels of their applications. When developers do not properly filter and sanitize user inputs, attackers can exploit this weakness to insert their own code, which is then executed in unsuspecting users' browsers. For instance, if an attacker inserts a payload like

```
<img\%20src=1\%20onerror=alert('hacked')>
```

into the parameters of a GET/POST request, it could trigger a pop-up dialog on the web page, demonstrating the successful exploitation of the vulnerability and the execution of the attacker's malicious code.

2) Components of XSS Payload: Though, the XSS attack vectors are diverse and numerous, are not randomly generated; they are meticulously crafted by attackers to exploit vulnerabilities in web applications. These attack vectors adhere to specific semantics and characteristics that make them effective in compromising the security of a website or web application. At their core, the essence of an XSS attack vector is to contain the malicious code that will be executed within a user's browser. This code can serve various malicious purposes, including stealing sensitive user information, hijacking user sessions, defacing web pages, or spreading malware.

One critical aspect of creating an effective XSS attack vector is ensuring that its structure aligns with the context of the output point within the web application. Attackers carefully consider where the payload will be injected, whether it's within a script tag, an HTML attribute, or a JavaScript function call. The attack vector must be tailored to the specific context to ensure that the injected code executes as intended. This contextual awareness is a key element that sets successful XSS attacks apart from failed attempts.

3) Payload Injection Location: XSS attacks can manifest in various contexts within web applications, each offering attackers different opportunities to exploit vulnerabilities. In this research, we explore several typical contexts where XSS attacks may occur, recognizing that this list is not comprehensive:

1) Inside HTML Tag Body: In this context, an attacker can inject malicious code directly into the body of an HTML element. For example:

   ```
   <p>...some text... <script>alert('XSS')
   ;</script></p>
   ```

2) HTML Tag Attribute Values: Attackers can also insert malicious code into the values of HTML tag attributes. For instance:

   ```
   <a href="javascript:alert('XSS')
   ">Click me</a>
   ```

3) Entire HTML Tag: An attacker can entirely replace or manipulate HTML tags to execute malicious actions. Here's an example:

   ```
   <img src="invalid-image" onerror="alert
   ('XSS')">
   ```

4) HTML Comments: Attackers might exploit HTML comments to hide malicious code. For example:

   ```
   <!-- <script>alert('XSS');</script> -->
   ```

5) JavaScript Context: In a JavaScript context, an attacker may inject malicious code that executes when a vulnerable web page is loaded. For example, an attacker could inject the following JavaScript code into a vulnerable input field:

   ```
   <img src='x' onerror='alert("XSS")'>
   ```

   When the web page containing this input field is loaded, the injected code within the `onerror` attribute of the `<img>` tag will trigger a pop-up alert with the message "XSS". This demonstrates how attackers can exploit JavaScript contexts to execute malicious code within web applications.

In essence, XSS attacks are not arbitrary; they are the result of a deliberate and strategic effort by attackers who understand the intricate details of web application security. Defenders must similarly be aware of these attack vectors and employ comprehensive security practices, including automatic payload generation, to identify and address these vulnerabilities effectively within their applications.

B. Large Language Models

Large Language Models (LLMs), represent a category of machine learning models proficient in processing and generating natural language text. These models commonly undergo training on extensive corpora of text data and leverage deep learning methodologies to grasp the intricacies of linguistic patterns and structures. The evolution of LLMs reached a significant milestone in 2017 with the advent of the Transformer model [9]. The Transformer's breakthrough capability included the ability to capture long-range linguistic dependencies effectively and enabled parallel training across multiple Graphics Processing Units (GPUs), thereby facilitating the training of substantially larger models.

These models acquire a deep understanding of the complex patterns and connections within the data, empowering them to generate fresh content mirroring the style and traits of

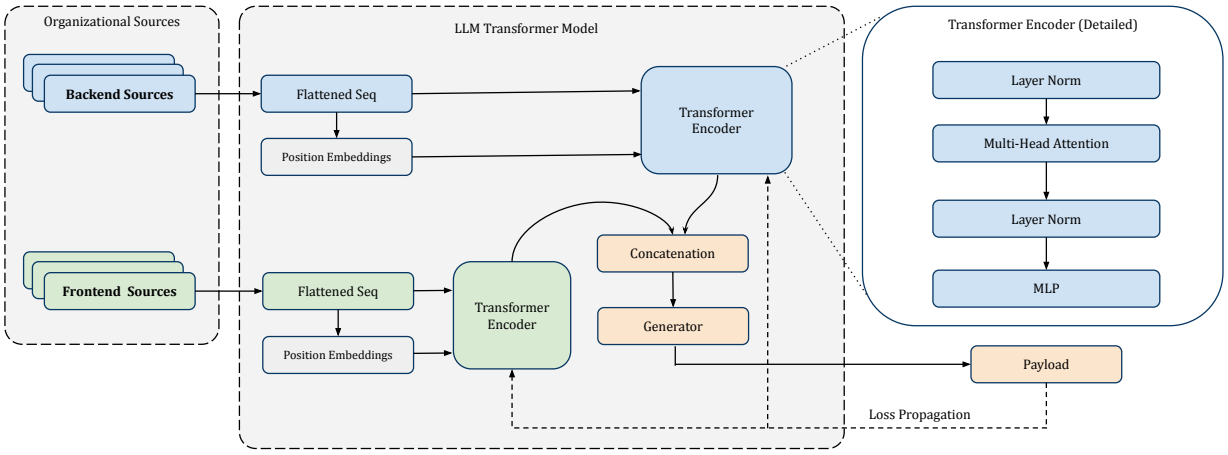

Fig. 1: Porposed Architecture of the LL-XSS Autoregressive Model

a particular author or genre. The process commences with pre-training, where the LLM is immersed in an extensive dataset comprising diverse text sources like books, articles, and websites. Employing unsupervised learning, the model predicts the subsequent word in a sentence by analyzing the contextual cues of preceding words. This process equips the model with a grasp of grammar, syntax, and semantic correlations, enabling it to produce coherent and contextually relevant text [10].

Transformers [9] represent the current cutting-edge deep learning models for processing real-world data using attention mechanisms in both generative and discriminative machine learning. These models serve as the basis for renowned deep learning models, including BERT [11], GPT-3 [12], DALL-E [13], and ChatGPT [14], which incorporate transformer modules and reinforcement learning.

Unlike the Convolutional Neural Networks (CNNs), transformer models employ self-attention mechanisms that enable the model to assign varying degrees of importance to different segments of the input sequence, depending on their relevance to the current task. This capacity enables the model to comprehend the context and interconnections among different segments of the input sequence, a critical factor for tasks like XSS payload generation [9]. The attention mechanism is essentially determined by the equation 1.

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V \tag{1}$$

The attention model is calculated by querying a specific feature of interest, represented by the query Q, against a collection of masks, represented by K. Possible keys that can be queried are represented by d_k, and division by the square root of this value yields a hard maximum. This is then processed through the softmax function to determine probabilities. Different values that keys K_i can assume are compiled in V. For a more comprehensive insight into the attention model, we recommend referring to the original transformer paper [9].

III. Proposed Architecture: LL-XSS

In this section, we provide an in-depth description of our novel approach for generating XSS payloads using auto-regressive models. The architecture of our proposed model is visually illustrated in Figure 1, which outlines the key components and their interactions.

The model receives inputs from two primary sources: frontend code and backend code. The frontend code comprises the HTML and JavaScript components of the entire website. It's worth noting that we omit CSS from this input as it is not pertinent to the XSS payload generation process.

The backend code, on the other hand, has the potential to encompass code written in various programming languages. However, for the purposes of our experiments, we have limited our focus to Node.js code.

To facilitate effective training of the auto-regressive model, we take a backend code repository as input and generate a tokenized version of the entire source code. To enhance the model's understanding of the input structure, we incorporate positional encodings before feeding it into the model. These positional encodings serve to provide context and help the model discern the relationships between different code elements and their positions within the source code. This preprocessing step is crucial in ensuring that the model can effectively learn and generate XSS payloads based on the input code.

The process continues as the flattened sequences, along with their respective positional encodings, are directed into a multi-attention head model. This model consists of several layer normalization steps and culminates in a final multi-layer perceptron. Notably, our model incorporates several innovative concepts, including the use of skip connections.

Skip connections play a pivotal role in our model by circumventing the issue of the vanishing gradient problem. This problem often hampers the learning process in large language models with extensive context. By facilitating the flow of gradients through the network, skip connections enable more effective learning and convergence. The transformer encoder

is responsible for creating an embedding of the input from the backend codebase. This embedding captures essential features and relationships within the code.

A similar procedure is executed for the frontend codebase, where the input is processed through its own transformer model. Essentially, the outputs of these two separate instances of the transformer models are concatenated to generate a combined embedding. This merged embedding exhibits twice the number of dimensions but, more importantly, encapsulates the intricate relationships between tokens in both the frontend and backend code. This rich representation enables the model to comprehend the connections and dependencies between code elements from different sources, laying the foundation for effective XSS payload generation.

The concatenated embedding resulting from the frontend and backend code is subsequently supplied as input to a custom generative model, which is specifically designed to generate XSS payloads as its output.

It is important to note that, initially, before any training takes place, the model's output lacks valid input and may not conform to desired syntax or functionality standards. During the training process, the loss is backpropagated not only through the generative part of the model but also to the frontend and backend transformers. Loss of the model is computed as follows:

$$\mathcal{L}(T, G) = \mathcal{E}_{x \sim p_{\text{data}}(x)}[\log(T(x))] + \mathcal{E}_{z \sim p_z(z)}[\log(1 - T(G(z)))]$$

Where $\mathcal{L}(T, G)$ represents the overall loss function. T is the transformer model and G is the generator network, $\mathcal{E}_{x \sim p_{\text{data}}(x)}$ represents the expectation over real data samples and $\mathcal{E}_{z \sim p_z(z)}$ represents the expectation over noise samples. This comprehensive feedback loop ensures that the entire model adapts and learns from the generated outputs.

After undergoing training for several epochs, the model exhibits the capability to generate not only syntactically valid XSS payloads but also offers valuable insights into the codebases being analyzed. These insights can be instrumental in identifying potential vulnerabilities and enhancing the overall security of web applications.

In the subsequent section, we present the details of our experiments and elaborate on the outcomes and outputs generated by the model during these experimental trials.

IV. Results and Discussions

In our experiments, we conducted our analysis using the well-known vulnerable web application called "OWASP Juice Shop". Juice Shop is renowned for harboring numerous known vulnerabilities, as it was intentionally designed to serve as a testing ground for uncovering security flaws within web applications.

For our experiments, we supplied the source code of Juice Shop, which encompasses both the backend logic and the frontend templates, as input to our model. Initially, prior to any training, the generator component of our model produced outputs that were often incoherent and, in many cases, syntactically incorrect. This behavior was in line with our expectations, given the untrained nature of the model.

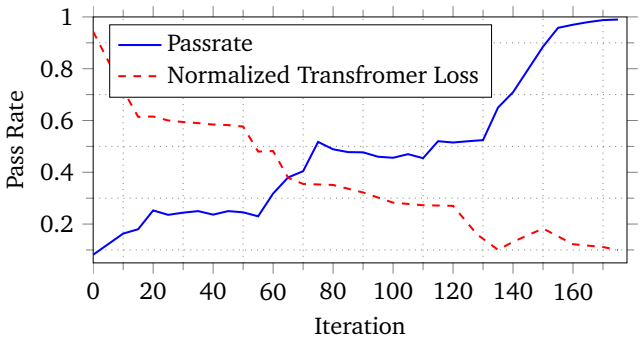

Fig. 2: Pass rate and loss plot

To quantitatively assess the quality of the payloads generated by our model, we devised a series of correctness tests. These tests were designed to parse the produced payloads and evaluate their validity, primarily focusing on whether they constituted valid JavaScript code. To achieve this, we employed the widely-used tool ESPRISMA, which specializes in tokenizing and parsing JavaScript (or formally ECMAScript) code. The use of ESPRISMA allowed us to systematically evaluate the generated payloads for syntactical correctness, contributing to the overall reliability and effectiveness of our model in producing meaningful XSS payloads.

After each training iteration, we generated precisely 100 payloads, and the payloads that were syntactically correctly formatted were marked as "pass". The pass rate for each iteration was calculated and is reported in Figure 2. This measure of functional correctness serves as a standard metric that has been employed in previous research to evaluate the accuracy of code generation from large language models. Table I details some of the correctly formated payload generated by our model.

While this approach provides us with a quantitative gauge of syntactic correctness in the generated code, the effectiveness of the payloads themselves was assessed manually. For this evaluation, we analyzed the payloads generated at the conclusion of the training and manually extracted payloads, along with the endpoints, that successfully identified vulnerabilities within the Juice Shop application.

Our analysis led to the discovery of reflective XSS attacks and stored XSS attack. However, it's important to note that, with our current model and training methodology, we were unable to identify DOM-based XSS vulnerabilities, despite the presence of known vulnerabilities of this type within the Juice Shop application. We speculate that this limitation may be attributed to our model's lack of round-trip integration tests. To potentially address this limitation, we envision incorporating a comprehensive quality assurance (QA) pipeline into our model, which may enhance our ability to detect DOM-based XSS vulnerabilities.

XSS Context	LL-XXS generated payload
Inside HTML Tag Body	
1. Image Source (data URI):	`<img src="data:image/svg+xml,<svg/onload=alert('XSS')>" />`
HTML Tag Attribute Values	
2. Anchor Tag Href Attribute (data URI):	`<a href="data:text/html,<img%20src=x%20onerror=alert('XSS') >">Click me</a>`
HTML Tag	
3. IFrame Tag (data URI):	`<iframe src="data:text/html,<img src=x onerror=alert('XSS')> "></iframe>`
HTML Comments	
4. HTML Comment with Data URI:	`<!-- <img src="data:text/html,<img src=x onerror=alert('XSS')> "> -->`
JavaScript	
5. Script Tag (data URI):	`<script src="data:text/javascript,alert('XSS')"></script>`

TABLE I: Examples for LL-XXS Generated Payload

This exciting avenue for future research represents an important direction for our work, as it holds the potential to further refine and strengthen our approach to XSS vulnerability detection and mitigation.

A. LL-XSS generated payload

The LL-XSS generated payloads target different contexts and endpoints within the Juice Shop application. For unstance, the reflected XSS exploits links with payloads in anchor tag href attributes. While the persistent XSS injects code into input fields, styles, and more, compromising user data. These examples showcase the diverse attack vectors for executing JavaScript within the experimental web application. Below we discuss some representitive indvidual samples of payload generated by our model.

1) **Image Source (data URI):**
The first payload given in Table I utilizes a data URI scheme within an `<img>` tag's `src` attribute. The data URI contains an SVG image with an `onload` event that triggers the `alert('XSS')` JavaScript code when the image loads. Attackers can use this technique to execute malicious code when the image is rendered.
2) **Anchor Tag Href Attribute (data URI):**
Paylod 2 showcases a data URI within the `href` attribute of an anchor `<a>` tag. The data URI contains an HTML snippet with an `<img>` tag that includes an `onerror` event to trigger the `alert('XSS')` code when the link is clicked.
3) **IFrame Tag (data URI):**
Payload 3 in the list, utilizes an `<iframe>` element with a data URI in the `src` attribute. The data URI encodes an HTML snippet with an `<img>` tag and an `onerror` event, executing the `alert('XSS')` code within the frame.
4) **HTML Comment with Data URI:**
The LL-XSS also generated a payload (payload 4) that uses an HTML comment to conceal a data URI containing malicious code. When the comment is processed, the code within the data URI executes, potentially leading to security vulnerabilities.
5) **Script Tag (data URI):**
Payload 5 uses one of the most obviuos technique used in XSS attacks. A `<script>` tag's `src` attribute is used with a data URI to load JavaScript code that triggers the `alert('XSS')` when the script is executed. This technique allows attackers to inject and execute JavaScript code.

V. Related Work

In the domain of injection vulnerability analysis, conventional approaches, as evidenced in various studies focusing on XSS [15], SQLi [16], and XML injection [17] predominantly depend on the use of pre-existing payloads. These methods aim to detect vulnerabilities by employing known malicious patterns. In contrast, the proposed LL-XSS stands out as it follows a dynamic and proactive approach, placing a lesser emphasis on prior payload data.

Studies in [18] and [19], leverage dynamic data analysis to identify potentially vulnerable input parameters. Similarly, researchers in [20], [21] systematically extract injection points and rigorously test them with known XSS attack strings. Furthermore, [22] introduces a novel approach, using custom browsers for taint tracking to create escape strings for addressing DOM-based XSS vulnerabilities.

Trickel et al. introduced a semi-automatic fuzzing scheme [23], but it involves intensive manual log and response analysis. Garn et al. presented an attack grammar mode with constraints to enhance web security testing [24]. Van Rooij

et al. introduced a fuzzy reasoning-based fuzzing method for discovering XSS vulnerabilities, primarily effective for reflective XSS [25]. Muhammad et al. [26] used vulnerability features for automated black-box web vulnerability scanning. Melicher et al. employed taint tracking to effectively detect DOM-based XSS vulnerabilities [5], demonstrating its efficacy in experiments. The proposed LL-XSS model distinguishes itself with its dynamic and proactive approach, enhancing its effectiveness and adeptness in addressing emerging vulnerabilities.

VI. Conclusion

In conclusion, our research presents a novel approach to detecting XSS vulnerabilities in web applications, with a specific focus on the context of automated XSS payload generation. Leveraging generative AI, our model swiftly generates malicious scripts that expose potential vulnerabilities identified during analysis, greatly expediting the identification process compared to manual methods. This advancement is particularly valuable as demonstrated by the diverse payload examples, revealing various attack scenarios. Our model's strength lies in its ability to generate subtle malicious scripts that evade traditional security measures. This enables accurate vulnerability detection and empowers security professionals to fortify web application security, reducing real-world attack risks.

References

[1] G. E. Rodríguez, J. G. Torres, P. Flores, and D. E. Benavides, "Cross-site scripting (xss) attacks and mitigation: A survey," *Computer Networks*, vol. 166, p. 106960, 2020.

[2] S. Gupta and B. B. Gupta, "Cross-site scripting (xss) attacks and defense mechanisms: classification and state-of-the-art," *International Journal of System Assurance Engineering and Management*, vol. 8, pp. 512–530, 2017.

[3] H. Hanif, M. H. N. M. Nasir, M. F. Ab Razak, A. Firdaus, and N. B. Anuar, "The rise of software vulnerability: Taxonomy of software vulnerabilities detection and machine learning approaches," *Journal of Network and Computer Applications*, vol. 179, p. 103009, 2021.

[4] X. Chen, M. Li, Y. Jiang, and Y. Sun, "A comparison of machine learning algorithms for detecting xss attacks," in *Artificial Intelligence and Security: 5th International Conference, ICAIS 2019, New York, NY, USA, July 26–28, 2019, Proceedings, Part IV 5*. Springer, 2019, pp. 214–224.

[5] W. Melicher, C. Fung, L. Bauer, and L. Jia, "Towards a lightweight, hybrid approach for detecting dom xss vulnerabilities with machine learning," in *Proceedings of the Web Conference 2021*, 2021, pp. 2684–2695.

[6] B. Gogoi, T. Ahmed, and H. K. Saikia, "Detection of xss attacks in web applications: A machine learning approach," *International Journal of Innovative Research in Computer Science & Technology (IJIRCST) ISSN*, pp. 2347–5552, 2021.

[7] Z. Liu, Y. Fang, C. Huang, and Y. Xu, "Gaxss: effective payload generation method to detect xss vulnerabilities based on genetic algorithm," *Security and Communication Networks*, vol. 2022, pp. 1–15, 2022.

[8] M. Foley and S. Maffeis, "Haxss: Hierarchical reinforcement learning for xss payload generation," in *2022 IEEE International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*. IEEE, 2022, pp. 147–158.

[9] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," *Advances in Neural Information Processing Systems*, vol. 30, 2017.

[10] C. Zhu, W. Ping, C. Xiao, M. Shoeybi, T. Goldstein, A. Anandkumar, and B. Catanzaro, "Long-short transformer: Efficient transformers for language and vision," *Advances in neural information processing systems*, vol. 34, pp. 17 723–17 736, 2021.

[11] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.

[12] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell *et al.*, "Language models are few-shot learners," *Advances in Neural Information Processing Systems*, vol. 33, pp. 1877–1901, 2020.

[13] A. Ramesh, M. Pavlov, G. Goh, S. Gray, C. Voss, A. Radford, M. Chen, and I. Sutskever, "Zero-shot text-to-image generation," in *International Conference on Machine Learning*. PMLR, 2021, pp. 8821–8831.

[14] OpenAI, "Chatgpt: Optimizing language models for dialogue," https://chat.openai.com/chat, 2023.

[15] K. Hasegawa, S. Hidano, and K. Fukushima, "Automating xss vulnerability testing using reinforcement learning." in *Proceedings of the 9th International Conference on Information Systems Security and Privacy(ICISSP 2023)*, 2023, pp. 70–80.

[16] M. S. Aliero, I. Ghani, K. N. Qureshi, and M. F. Rohani, "An algorithm for detecting sql injection vulnerability using black-box testing," *Journal of Ambient Intelligence and Humanized Computing*, vol. 11, pp. 249–266, 2020.

[17] S. Jan, A. Panichella, A. Arcuri, and L. Briand, "Automatic generation of tests to exploit xml injection vulnerabilities in web applications," *IEEE Transactions on Software Engineering*, vol. 45, no. 4, pp. 335–362, 2019.

[18] A. Alhuzali, R. Gjomemo, B. Eshete, and V. Venkatakrishnan, "{NAVEX}: Precise and scalable exploit generation for dynamic web applications," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018, pp. 377–392.

[19] S. Bensalim, D. Klein, T. Barber, and M. Johns, "Talking about my generation: Targeted dom-based xss exploit generation using dynamic data flow analysis," in *Proceedings of the 14th European Workshop on Systems Security*, 2021, pp. 27–33.

[20] M. Mohammadi, B. Chu, H. R. Lipford, and E. Murphy-Hill, "Automatic web security unit testing: Xss vulnerability detection," in *Proceedings of the 11th International Workshop on Automation of Software Test*, 2016, pp. 78–84.

[21] S. Gupta and B. B. Gupta, "Robust injection point-based framework for modern applications against xss vulnerabilities in online social networks," *International Journal of Information and Computer Security*, vol. 10, no. 2-3, pp. 170–200, 2018.

[22] S. Lekies, B. Stock, and M. Johns, "25 million flows later: large-scale detection of dom-based xss," in *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, 2013, pp. 1193–1204.

[23] E. Trickel, F. Pagani, C. Zhu, L. Dresel, G. Vigna, C. Kruegel, R. Wang, T. Bao, Y. Shoshitaishvili, and A. Doupé, "Toss a fault to your witcher: Applying grey-box coverage-guided mutational fuzzing to detect sql and command injection vulnerabilities," in *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2023, pp. 2658–2675.

[24] B. Garn, J. Zivanovic, M. Leithner, and D. E. Simos, "Combinatorial methods for dynamic gray-box sql injection testing," *Software Testing, Verification and Reliability*, vol. 32, no. 6, p. e1826, 2022.

[25] O. van Rooij, M. A. Charalambous, D. Kaizer, M. Papaevripides, and E. Athanasopoulos, "webfuzz: Grey-box fuzzing for web applications," in *Computer Security–ESORICS 2021: 26th European Symposium on Research in Computer Security, Darmstadt, Germany, October 4–8, 2021, Proceedings, Part I 26*. Springer, 2021, pp. 152–172.

[26] M. Mohammadi, B. Chu, and H. R. Lipford, "Detecting cross-site scripting vulnerabilities through automated unit testing," in *2017 IEEE International Conference on Software Quality, Reliability and Security (QRS)*. IEEE, 2017, pp. 364–373.