Automated Test Input Generation Based on Web User Interfaces via Large Language Models

Kento Hasegawa[1,*][a], Hibiki Nakanishi[2,*], Seira Hidano[1], Kazuhide Fukushima[1][b],
Kazuo Hashimoto[2] and Nozomu Togawa[2]

[1]*KDDI Research, Inc., 2-1-15, Ohara, Fujimino-shi, Saitama, Japan*
[2]*Waseda University, 3-4-1, Okubo, Shinjuku-ku, Tokyo, Japan*

Keywords: Internet of Things, Cybersecurity, Large Language Models, Fuzzing, User Interfaces.

Abstract: The detailed implementation of IoT devices is often opaque, necessitating the use of a black-box model for verification. A challenge in fuzzing for the diverse types of IoT devices is generating initial test inputs (i.e., initial seeds for fuzzing) that fit the specific functions of the target. In this paper, we propose an automatic test input generation method for fuzzing the management interfaces of IoT devices. First, the automated web UI navigation function identifies the input fields. Next, the test input generation function creates appropriate test inputs for these input fields by analyzing the surrounding information of each field. By leveraging these functions, we establish a method for automatically generating test inputs specifically for the web user interfaces of IoT devices. The experimental results demonstrate that test inputs that are suitable for the input fields are successfully generated.

1 INTRODUCTION

Due to the proliferation of IoT devices, such devices are being used in various aspects of our daily lives. In addition, the number of concerns about the impact of cybersecurity on IoT devices has increased. Thus, enhancing the cybersecurity of IoT devices is a critically important challenge.

For users of IoT devices, verifying the security of these products can be challenging for several reasons. First, some IoT devices have firmware that is encrypted or otherwise protected, making it difficult to examine binary data. Second, owing to the wide variety of functions that IoT devices offer, it is challenging to prepare a standardized verification process. In summary, the following points present challenges in security verification for IoT devices:

- The detailed implementation of IoT devices is often opaque, necessitating the use of a black-box model for verification.

- The diverse range of functions requires corresponding tailored verifications.

One approach to performing security verification under a black-box model is fuzzing. In fuzzing, inputs that are likely to cause malfunctions are continuously fed to the IoT device under inspection. By checking whether any issues actually arise based on the device's responses, fuzzing helps in discovering unknown vulnerabilities. Since fuzzing examines the device by verifying the responses to inputs, it is a promising approach for inspecting IoT devices (Eceiza et al., 2021).

Generally, fuzzing involves seed generation, which determines the initial input, and mutation, which alters the seed. A challenge in fuzzing for the diverse types of IoT devices is generating initial test inputs (i.e., initial seeds) that fit the specific functions of the target. For example, if a field only accepts numerical input, then providing a string will produce an error, assuming that the input validation functions properly reject the input. If string data are generated during seed generation, then subsequent mutations will continue to produce string data, making the tests less efficient. Therefore, seed generation is crucial.

In the input and output of IoT devices, the management interface is particularly vulnerable. This is because it often allows access from external sources via the internet; among the various functions of IoT devices, the ability to be accessed through a web browser is a common feature. By focusing on the web user interface (UI), we propose a fuzzing tool that can be applied to a broader range of IoT devices.

---

[a] https://orcid.org/0000-0002-6517-1703

[b] https://orcid.org/0000-0003-2571-0116

*Kento Hasegawa and Hibiki Nakanishi contributed equally to this work.

In this paper, we propose an automatic test input generation method for fuzzing the management interfaces of IoT devices. The proposed method consists of two main functions, namely, automated web UI navigation and test input generation. The automated web UI navigation function identifies the input fields that need to be tested based on the content displayed on the web UI. The test input generation function creates appropriate test inputs for these input fields by analyzing the surrounding information of each field. By leveraging these functions, we establish a method for automatically generating test inputs specifically for the web UIs of IoT devices.

The contributions of this paper can be summarized as follows:

- We propose a framework that automates the navigation of the web UI and uses a large language model (LLM) to generate test inputs. By including information about the target input fields and the names of the vulnerabilities to be tested within the prompt, test inputs can be generated efficiently to examine those vulnerabilities.

- By employing chain-of-thought (CoT) and error history referencing, we enhance the efficiency of test input generation. Specifically, test inputs can be modified based on input validation error messages, thereby enabling the generation of test inputs that are suitable for the input fields of IoT devices.

2 BACKGROUND

2.1 Fuzzing on IoT Devices

Fuzzing is a method used for discovering vulnerabilities in devices by continuously providing input that may cause malfunctions and checking the device's responses to these inputs. If the device exhibits unintended behavior, such as freezing, then the input at that time is considered to have caused a fault. Fuzzing can be applied via either a white-box or black-box approach, depending on whether the internal information of the target device is accessible. In the case of IoT devices, it is often difficult for users to access their internal logic because the firmware is frequently encrypted and protected. Therefore, a black-box testing approach is typically adopted.

Fuzzing for IoT devices can be classified into three categories (You et al., 2022), namely, firmware-based, companion-app-based, and network-based methods. The firmware-based method (Zheng et al., 2019; Kim et al., 2020) generates test inputs based on analysis of the firmware. However, as mentioned earlier, the applicability of this method is limited because many IoT devices have protected firmware. The companion-app-based method (Chen et al., 2018; Redini et al., 2021) analyzes applications used to manage IoT devices to recognize control commands, generating test inputs based on these commands. Analyzing companion apps (e.g., smartphone applications) is a more feasible approach than analyzing firmware. However, several IoT devices do not have such applications available, making this method inapplicable in those cases. The network-based method (Song et al., 2019; Feng et al., 2021) captures and analyzes the communication data sent and received by the IoT device to generate test inputs based on this communication. This approach allows the easy analysis of the input and output content of IoT devices by examining the communication through a proxy, as well as the communication with the web UI via the web browser.

2.2 Network-Based Fuzzing

In the network-based approach, fuzzing is conducted by capturing and analyzing the communications of IoT devices. Typically, IoT devices communicate with external devices via specific protocols or syntax; communications that do not adhere to these rules are typically treated as errors. Therefore, test inputs that differ substantially from the device's protocol or syntax are immediately processed as errors, rendering vulnerability testing ineffective. Thus, extracting the protocol and syntax from the communication content is crucial. However, preparing detailed descriptions of protocols and syntax used across different IoT devices for the test phase entails a significant workload. The network-based method is characterized by its ability to generate test inputs by predicting protocols and syntax from actual communication content.

In Snipuzz (Feng et al., 2021), snippets of strings are extracted from captured communication content. By generating mutated test inputs based on the syntax of these extracted snippets, it is possible to align with the communication protocols and syntax of IoT devices.

However, the network-based method has drawbacks. It is not effective against communications that have been secured. For example, some manufacturers' IoT devices add a hash value based on the current time to commands sent from the web UI to the IoT device. The device verifies that the command is correct by comparing the command with the hash value. Figures 1 and 2 show examples of such communication logs. In Figure 1, the characters highlighted in red indicate the hash value calculated based on the XML

content presented in Figure 2. In Figure 2, the characters highlighted in red indicate the hash value based on the input password. On the other hand, Figure 3 shows an example where test inputs are generated by fuzzing. The parts highlighted in red indicate the segment that is generated through the fuzzing process. If the content in the "LoginPassword" tag is modified, the hash value in the HTTP header (as shown in red color in Figure 1) must be changed. However, in the existing fuzzing process, it is difficult to determine appropriate values for such an HTTP header. Therefore, directly modifying the captured communication content causes a mismatch with the hash value that is calculated based on the original communication content, making it impossible to apply existing methods to IoT devices with such features.

An approach can be considered where test strings are provided to input fields in the web UI, allowing for direct manipulation to conduct fuzzing. Zhang et al. proposed a fuzzing method that targets web UIs (Zhang et al., 2021). This method conducts fuzzing based on state transitions by analyzing the authentication mechanisms employed in the web UI. However, actual IoT devices are equipped with a variety of authentication mechanisms. Considering this, there remain several challenges for fuzzing that targets a diverse range of IoT devices.

- It is necessary to test for vulnerabilities that are common in general web applications, such as cross-site scripting, as well as vulnerabilities, such as buffer overflow.

- IoT devices with various functionalities have diverse input rules, necessitating the generation of test inputs tailored to each field.

Especially in fuzzing, the generation of initial inputs, as previously discussed, poses a problem. Similar to the approach employed in the network-based fuzzing, generating initial inputs based on the context used in the target device is a reasonable approach. However, since the applications of IoT devices vary, it is difficult to generate initial inputs that fit to the target input field. LLMs can solve the problem by interrupting the context of the input fields and generating texts based on the context. In this paper, we propose a method for generating test inputs based on the web UI of IoT devices using LLMs.

3 PROPOSED METHOD

In this paper, we propose an automatic test input generation method for fuzzing the web-based management interfaces of IoT devices. Figure 4 shows an

```
POST /HNAP1/ HTTP/1.1
Host: 192.168.XXX.YYY
...
X−Requested−With: XMLHttpRequest
HNAP_AUTH:
    A1DBEC474B2C71619E38D09E0E96542B
    1181621715
...
```

Figure 1: Example of the HTTP header in a UI-based communication (partially omitted).

```
<?xml version="1.0" encoding="utf-8"?>
  <soap:Envelope ...>
    <soap:Body>
      <Login ...>
        ...
        <LoginPassword>
          ECD57B4D4825870353074A658519BB2C
        </LoginPassword>
        <Captcha>
        </Captcha>
      </Login>
    </soap:Body>
  </soap:Envelope>
  ...
```

Figure 2: Example of a UI-based communication content (partially omitted).

overview of the proposed method.

The proposed method consists of the following four steps:

Step 1. **Input Fields Extraction:** Input fields are extracted from the web UI of the target IoT device.

Step 2. **Test Input Generation:** Test inputs are generated for each input field.

Step 3. **Error Handling:** Errors (e.g. input validation errors) are resolved by interpreting error messages.

Step 4. **Form Submission:** The form content is submitted to the target IoT device.

First, in Step 1, the input fields and their captions on the web UI are extracted via automated web browser control. In Step 2, test inputs are generated based on the extracted content. In Step 3, the generation of the test inputs is repeated until error messages disappear after the test inputs are filled in or the form is submitted. Once the error messages are resolved, test inputs based on the generated test inputs are submitted, and a response is obtained to verify the behavior of the web UI in Step 4.
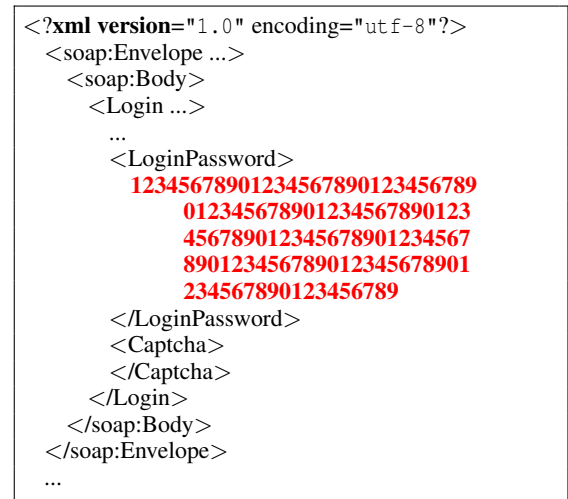
```
<?xml version="1.0" encoding="utf-8"?>
  <soap:Envelope ...>
    <soap:Body>
      <Login ...>
        ...
        <LoginPassword>
          123456789012345678901234567890123456789
          012345678901234567890123
          456789012345678901234567
          890123456789012345678901
          234567890123456789
        </LoginPassword>
        <Captcha>
        </Captcha>
      </Login>
    </soap:Body>
  </soap:Envelope>
  ...
```

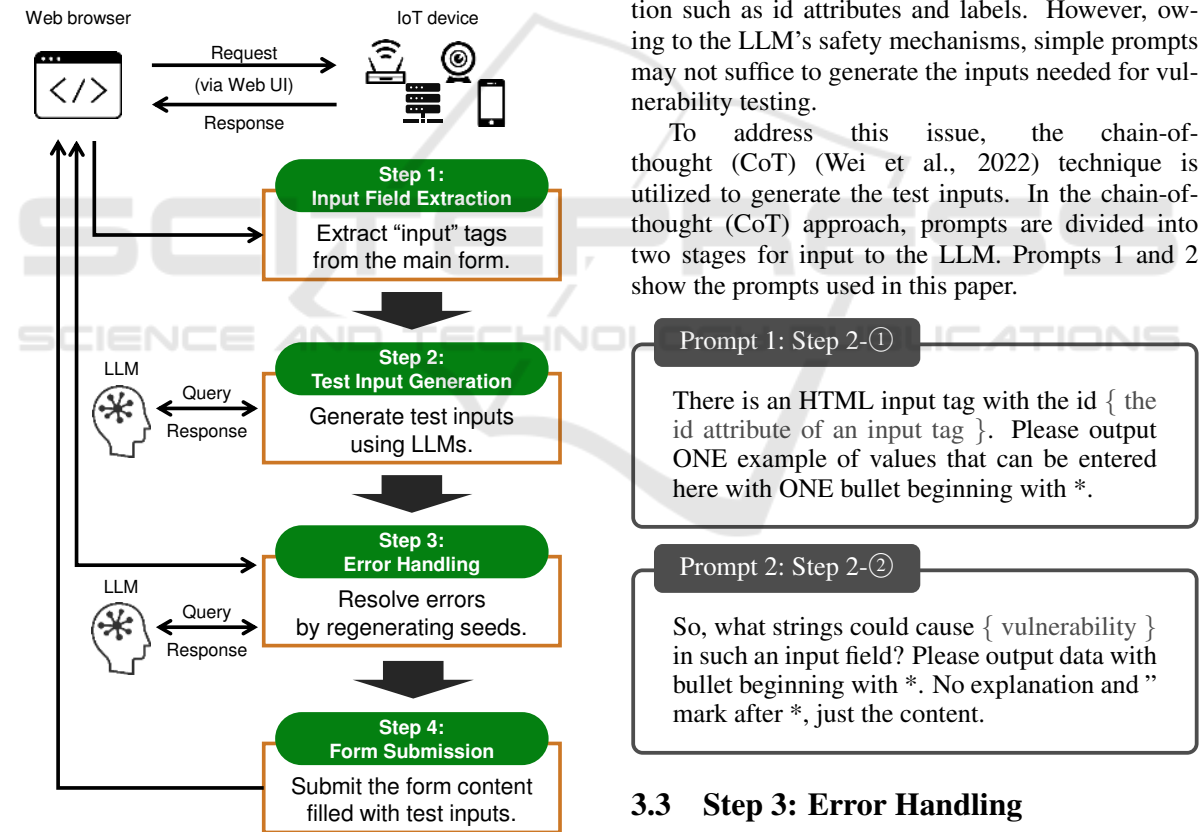Figure 3: Example of a mutated test input (partially omitted).

Figure 4: Overview of the proposed method.

3.1 Step 1: Input Fields Extraction

To automate the browser control, we use Playwright [a], which is a tool commonly used for testing web ap-

---

[a]https://github.com/microsoft/playwright-python

plications. Playwright allows for the control of web browsers and the retrieval of requests and responses from external programs.

First, the URL specified as the web management interface of the IoT device is opened. After the content is rendered, the "form" tags contained within the main content area of the rendered web page are extracted. Within these forms, we further extract input tags that are capable of receiving input and identify them as input fields. During this process, we obtain the "id" attribute to identify each input field. If the "id" attribute is not set, then other attributes, such as the "name" attribute, are retrieved instead.

3.2 Step 2: Test Input Generation

Based on the set of "input" tags obtained in Step 1, test inputs are generated. For each test input, initial values are created via an LLM, which uses information such as id attributes and labels. However, owing to the LLM's safety mechanisms, simple prompts may not suffice to generate the inputs needed for vulnerability testing.

To address this issue, the chain-of-thought (CoT) (Wei et al., 2022) technique is utilized to generate the test inputs. In the chain-of-thought (CoT) approach, prompts are divided into two stages for input to the LLM. Prompts 1 and 2 show the prompts used in this paper.

Prompt 1: Step 2-①

There is an HTML input tag with the id { the id attribute of an input tag }. Please output ONE example of values that can be entered here with ONE bullet beginning with *.

Prompt 2: Step 2-②

So, what strings could cause { vulnerability } in such an input field? Please output data with bullet beginning with *. No explanation and " mark after *, just the content.

3.3 Step 3: Error Handling

Input fields in a web UI typically have functionalities that validate the entered content. As a result, values that are entirely different from what the web UI anticipates are not sent to the IoT device. For example, in an IPv4 address input field, the expected format is a series of up to three-digit numbers concatenated by periods. If the input validation function outputs an

error, the entered content will not be sent to the IoT device, thus preventing fuzzing verification on the device's firmware. Therefore, it is essential to generate inputs that pass validation.

Prompts 3 and 4 show the prompts used in this paper.

> **Prompt 3: Step 3-①**
>
> The following error message was returned:{ the displayed error message }. What format should I use to avoid the error message?

> **Prompt 4: Step 3-②**
>
> Based on that, what string is likely to cause a { vulnerability } in this input field? If this field does not involve a { vulnerability }, generate another input value. Please output data with bullet beginning with *. No explanation and " mark after *, just the content.

3.4 Step 4: Form Submission

After all the errors are resolved, the form filled with generated texts is sent to the IoT device. When a form is submitted, a response is returned. By examining the contents of the returned response, it can be determined whether the IoT device functioned correctly or if an unintentional error occurred.

4 EXPERIMENTS

In the experiment, we evaluate the effectiveness of the proposed method through input testing targeting IoT devices. This paper focuses on the following research questions for evaluation:

- **RQ1:** Can the proposed method automatically generate test inputs?
- **RQ2:** How does the efficiency of test input generation change when different LLMs are used?

To answer to RQ1 and RQ2, we conducted Experiments 1 and 2.

4.1 Setup

Python 3.10 was used for the implementation. Playwright was employed to interact with the web UI. The LLMs used were Mistral-7B-Instruct-v0.2 (Jiang et al., 2023) and Vicuna-7B-v1.5 (Zheng et al., 2023).

The IoT devices targeted for testing were a router and an IP camera. These devices were operated in an emulation environment via the firmware emulator, FirmAE (Kim et al., 2020). Both devices can be controlled via their web UIs. In the experiment, test inputs are generated specifically for these web UIs.

4.2 Experiment 1: Test Value Generation

In Experiment 1, to address RQ1, we evaluated whether test inputs corresponding to various vulnerabilities could be generated for the web UIs of two different types of IoT devices. The vulnerabilities targeted for testing included null pointer exception, buffer overflow, and cross-site scripting (XSS). A null pointer exception may be caused by treating "null" vaules. A buffer overflow may be caused when the program receives a longer message than expected. XSS vulnerability may be caused by injecting scripts into the input field. We expect that the proposed method could generate test inputs automatically to validate such vulnerabilities.

Prompt 5, Response 1, Prompt 6, and Response 2 show a series of prompt and response examples obtained from the experiment that targeted the web UI of a router. In this experiment, we used the Mistral LLM. In Prompt 5, the LLM was instructed to infer the possible input formats based on the "id" attributes of the input fields. Next, in Prompt 6, the LLM was directed to generate test inputs that could be used to verify given vulnerabilities. Finally, as shown in Response 2, multiple test input candidates were successfully obtained.

> **Prompt 5: Step 2-① Example**
>
> There is an HTML input tag with the id #password PPTP. Please output examples of values that can be entered here.

> **Response 1: Step 2-① Example**
>
> A strong password with a minimum of 12 characters, including a mix of uppercase and lowercase letters, numbers, and special characters. For example: "Password123#Secure!"

> **Prompt 6: Step 2-② Example**
>
> So, what strings could cause XSS in such an input field? Please output data with bullet beginning with *.

Table 1: Generated test values using the proposed method.

Vulnerability	Target	#	Generated test value
Null Pointer Exception	Router	1	`null`
		2	`document.getElementById("#password PPTP").value = null;`
		3	`#maximumIdleTime PPTP=null`
	IP Camera	1	`null`
		2	`SSID.value = null`
		3	`name='PreSharedKey' = null`
Buffer Overflow	Router	1	`\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00 \x00\x00...`
		2	`1234567890123456789012345678901234567890123456789012345678901234 567890`
		3	`abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789`
	IP Camera	1	`PreSharedKeyA1B2C3D4E5F6G7H8I9J10K11L12M13N14O15P16Q17R18S19T20`
		2	`"A"*\*n\*n\*n\*n\*n\*n\*n\*n\*n\*n\*n\*n\*n\*n\*n\*n\*n\*n\*n\*n \*n\*n\*n\*n\*n\*n`
		3	`PreSharedKeyX1X2X3X4X5X6X7X8X9X10X11X12X13X14X15X16X17X18X19X20`
XSS	Router	1	`<script>alert("XSS Attack");</script>`
		2	`<meta http-equiv="refresh" content="0; url=malicious page.html">`
		3	`<base href=javascript:alert('XSS Attack')>`
	IP Camera	1	`<script>alert("XSS Attack!");</script>`
		2	`<meta http-equiv="refresh" content="0; url=malicious site.com">`
		3	`<img src="malicious-image.jpg" onerror="alert('XSS')">`

Response 2: Step 2-① Example

* <script>alert("XSS Attack!");</script>
* <imgsrc="javascript:alert('XSS Attack!');">
* <iframe src="javascript:alert('XSS Attack!') ;"></iframe>
* <object data="javascript:alert('XSS Attack!') ;"></object>
* <embed src="javascript:alert('XSS Attack!') ;">
* <style>bodybackground:url(javascript:alert(' XSS Attack!'));</style>
* <link rel="stylesheet" href="javascript:alert(' XSS Attack!');">

Table 1 presents examples of the results generated in Experiment 1. For two types of IoT devices, three test values for each of the three targeted vulnerabilities were generated. As shown in Table 1, a variety of test values were successfully produced, demonstrating the ability to generate diverse input types for different vulnerabilities.

4.3 Experiment 2

In Experiment 2, to address RQ2, the performances of two LLMs, namely, Mistral and Vicuna, were compared. Specifically, the comparison focused on error handling and test input generation capabilities.

Figure 5: The web UI when the input values are acceptable.

4.3.1 Comparison of Error Handling

In the web UI of IoT devices, if a text that is different from the expected input format is entered, an error message may be displayed, and the input may not be allowed to be sent to the IoT device.

Figure 5 shows an example of a web UI. When an unexpected text is entered into the input field, an error message appears. Figure 6 shows an example of such an error message. Although the mechanism for displaying errors varies by IoT device, such input validation is often handled by JavaScript in the web browser. In such cases, the input is not sent to the IoT device, preventing fuzzing on the device.

To address this issue, the proposed method provides a solution in Step 3, where error handling is implemented to overcome this limitation and enable fuzzing on the IoT device.

Table 2 presents the results of handling errors for the screen shown in Figure 6. In Table 2, #1 corresponds to the error message displayed for the "LAN IP Address" field, #2 corresponds to the "Sub-

Table 2: Example of handling errors.

#	Error message	Model		Generated test value
1	Please enter a valid IP address. (e.g. 192.168.0.1)	Mistral	Before:	`document.getElementById("lanIP") is null`
			After:	`172.16.254.1`
		Vicuna	Before:	`Empty string`
			After:	`192.168.0.1`
2	Please enter a valid IP address. (e.g. 255.255.255.0)	Mistral	Before:	`203.0.113.12`
			After:	`255.255.255.0`
		Vicuna	Before:	`Empty string ()`
			After:	`255.255.255.0`
3	Text field contains illegal characters.	Mistral	Before:	`document.getElementById("presentDeviceName")`
			After:	`John Doe`
		Vicuna	Before:	`Empty string ()`
			After:	`␣ (One space)`

Figure 6: The web UI when the input values cannot be accepted.

net Mask" field, and #3 corresponds to the "Management Link" field.

Initially, the LLM generated the test inputs shown in the "Before" row. In the initial step, as part of Step 2 and following the instructions outlined in Prompt 2, the LLM generated test inputs intended to test vulnerabilities such as the null pointer exception, which resulted in inputs that did not conform to the expected format of the input fields.

However, based on the error messages received, the LLM was further instructed through Prompts 3 and 4 to adjust the test inputs, leading to the generation of inputs shown in the "After" row. These adjusted inputs were in formats considered acceptable to the web UI, allowing the system to conduct further testing without encountering validation errors.

A comparison of the Mistral and Vicuna models revealed that both models were capable of interpreting the content of error messages and generating test inputs that were aligned with the error descriptions. However, the Vicuna model often produces outputs that exactly match the example values provided in the error messages. Further discussion on the variation in the test values, including this observation, will be provided in the following experiment.

Table 3: Comparison in the variation of generated test values.

Model	# of Test Examples
Mistral	3.125
Vicuna	2.25

4.3.2 Comparison in Test Generation

We compare the number of variations in generated test inputs. We examine the average number of test inputs generated when providing the prompt shown in Prompt 2 or Prompt 4 to the LLM.

Table 3 shows the average number of test inputs generated when Prompt 2 or Prompt 4 was attempted 8 times. As shown in Table 3, the Mistral model was able to generate more test inputs per prompt than the Vicuna model. Notably, while the number of test inputs generated may vary depending on the model used, the proposed method functions effectively regardless of the model employed.

5 DISCUSSION

5.1 Limitation

Identifying input fields and submitting buttons are challenging tasks. In the proposed method, input elements and buttons within a specific form tag are used. This process involves some manual effort.

However, actual IoT devices can have complex UIs, making the identification of input fields and submitting buttons difficult. To address this issue, the use of LLMs to analyze the structure of the HTML is suggested. Since processing the entire HTML at once is challenging due to token limitations, focusing the analysis on key parts is expected to assist in identifying input fields and submitting buttons effectively.

5.2 Future Work

In this paper, we propose a method targeting web UIs, with a focus on the generation of initial test inputs for fuzzing. This approach can be combined with existing black-box-based fuzzing techniques.

Examples of black-box-based fuzzing techniques include FuzzSim (Woo et al., 2013) and IoT Fuzzer (Chen et al., 2018). By using the proposed method as a basis, these existing black-box-based fuzzing techniques can mutate the given test inputs to continuously generate new test inputs. This approach enables the verification of even more vulnerabilities. Thus, the method proposed in this paper is useful in that it can be integrated with existing mutation techniques to increase the effectiveness of fuzzing efforts.

6 CONCLUSION

In this paper, we propose an automatic test input generation method for fuzzing the management interfaces of IoT devices. In the proposed method, the automated web UI navigation function identifies the input fields. The test input generation function creates appropriate test inputs by analyzing the surrounding information of each input field. By leveraging these functions, we establish a method for automatically generating test inputs specifically for the web UIs of IoT devices. Furthermore, the proposed method revises the generated test inputs by interpreting error messages displayed in the web UI. The experimental results demonstrate that test inputs that are suitable for the input fields are successfully generated. Future work will include the efficient mutation of the test input for fuzzing.

ACKNOWLEDGEMENTS

REFERENCES

Chen, J., Diao, W., Zhao, Q., Zuo, C., Lin, Z., Wang, X., Lau, W. C., Sun, M., Yang, R., and Zhang, K. (2018). Iotfuzzer: Discovering memory corruptions in iot through app-based fuzzing. In *25th Annual Network and Distributed System Security Symposium, NDSS*.

Eceiza, M., Flores, J. L., and Iturbe, M. (2021). Fuzzing the internet of things: A review on the techniques and challenges for efficient vulnerability discovery in embedded systems. *IEEE Internet of Things Journal*, 8(13):10390–10411.

Feng, X., Sun, R., Zhu, X., Xue, M., Wen, S., Liu, D., Nepal, S., and Xiang, Y. (2021). Snipuzz: Black-box fuzzing of iot firmware via message snippet inference. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, page 337–350.

Jiang, A. Q., Sablayrolles, A., Mensch, A., Bamford, C., Chaplot, D. S., de las Casas, D., Bressand, F., Lengyel, G., Lample, G., Saulnier, L., Lavaud, L. R., Lachaux, M.-A., Stock, P., Scao, T. L., Lavril, T., Wang, T., Lacroix, T., and Sayed, W. E. (2023). Mistral 7b.

Kim, M., Kim, D., Kim, E., Kim, S., Jang, Y., and Kim, Y. (2020). Firmae: Towards large-scale emulation of iot firmware for dynamic analysis. In *Proceedings of the 36th Annual Computer Security Applications Conference*, page 733–745.

Redini, N., Continella, A., Das, D., De Pasquale, G., Spahn, N., Machiry, A., Bianchi, A., Kruegel, C., and Vigna, G. (2021). Diane: Identifying fuzzing triggers in apps to generate under-constrained inputs for iot devices. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 484–500.

Song, C., Yu, B., Zhou, X., and Yang, Q. (2019). Spfuzz: A hierarchical scheduling framework for stateful network protocol fuzzing. *IEEE Access*, 7:18490–18499.

Wei, J., Wang, X., Schuurmans, D., Bosma, M., Xia, F., Chi, E., Le, Q. V., Zhou, D., et al. (2022). Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837.

Woo, M., Cha, S. K., Gottlieb, S., and Brumley, D. (2013). Scheduling black-box mutational fuzzing. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, page 511–522. Association for Computing Machinery.

You, M., Kim, Y., Kim, J., Seo, M., Son, S., Shin, S., and Lee, S. (2022). Fuzzdocs: An automated security evaluation framework for iot. *IEEE Access*, 10:102406–102420.

Zhang, H., Lu, K., Zhou, X., Yin, Q., Wang, P., and Yue, T. (2021). Siotfuzzer: Fuzzing web interface in iot firmware via stateful message generation. *Applied Sciences*, 11(7).

Zheng, L., Chiang, W.-L., Sheng, Y., Zhuang, S., Wu, Z., Zhuang, Y., Lin, Z., Li, Z., Li, D., Xing, E. P., Zhang, H., Gonzalez, J. E., and Stoica, I. (2023). Judging llm-as-a-judge with mt-bench and chatbot arena.

Zheng, Y., Davanian, A., Yin, H., Song, C., Zhu, H., and Sun, L. (2019). FIRM-AFL: High-Throughput greybox fuzzing of IoT firmware via augmented process emulation. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1099–1114.