GUI Smart Fuzzing on Modern Web Frameworks

Gabriel Gladstone
Computer Engineering
University of Virginia
Charlottesville, VA
zwy7ce@virginia.edu

Kyle Durrer
Computer Science
University of Virginia
Charlottesville, VA
kmd2zjw@virginia.edu

***Abstract*—It's common knowledge that all major software technologies require a significant amount of testing. However, less attention is given to the graphical user interfaces (GUI) of these software tools. In this work, we wish to extend the Fuzzing Book's GUI Ripper tool, used for state exploration for GUIs, to support modern web frameworks. We strategically pick specfic HTML attributes to support, backed by web scraping in-production GUIs of the most popular websites on the internet. We then evaluate our extensions and see if an LLM can help guide smart fuzzing inputs to reach deeper states in the GUI state graph. Performing an analysis of the resulting state graphs and outputs, we found that the results indicate our extensions made steps towards reaching deeper states for these modern GUIs. Our code is provided here:**

I. Introduction

In the modern software development life cycle, software testing demands a significant amount of attention. Graphical User Interfaces (GUIs) are just one element of many software architectures, but can be incredibly challenging to manually test exhaustively. The problem with GUI functional testing is that these user interfaces have an incredible amount of variables that include complexities like overlapping widgets, different devices like PC vs. mobile, and external factors like third-party APIs. Furthermore, even if there exists a test suite that adequately satisfies these conditions, the speed at which these GUIs are updated and tuned can make it difficult to maintain the test suite.

While there are many techniques to aid developers in GUI testing, like visual regression testing and recording browser activity for replayability, our work is interested in a technique called GUI Ripping. This advanced technique aims to automatically generate test suites by analyzing the GUI, extracting the widgets, and then creating the test suite from there. Using this idea, we aim to extend the Fuzzing Book's GUI Fuzzer that utilize GUI Ripping to support modern web frameworks like Bootstrap and Django [4]. Then we wish to incorporate an LLM to help guide the fuzzing tool to deeper program states, by utilizing the context of the web page to pick smarter inputs.

II. Related Works

Some simple GUI testing tools, like Selenium and Cypress, directly leverage the Browser's JavaScript events on the webpage to construct GUI program state. Intuitively, these tools are effectively functional unit tests that are easy to run but hard to develop since the developer has to manually create the inputs that make up the program's end state. Once these events are recorded through manual GUI navigation, they can easily be replayed to ensure that the GUI is still functional.

This manual process can be hard to create and maintain, so one major leap forward is to automatically create these test suites through GUI Ripping. In one prior work, Eggplant DAI utilizes traditional machine learning models to automatically create these unit tests. Although this technology appears pretty seamless, and even reports GUI coverage, there is still significant harnessing required to define what counts as a state in the specific application under test (AUT). Another prior work of interest is GPTDriod, an LLM agent that can take control of a Android GUI [3]. While this work leverages the power of LLMs and applies them to functional GUI testing, it doesn't utilize fuzzing techniques to create test suites, which is the focus of our work.

III. Methodology

To build a smart fuzzing tool to support functional testing of the modern web, approach this project as follows.

1) **Research established software**
   a) We research GUI Fuzzing tools to extend for our use case.
2) **Determine objectives.**
   a) We explore the modern landscape to determine where to best put our efforts.
3) **Build.**
   a) We add a variety of features both to reach our objectives and better explore the GUI landscape.
4) **Quantify**
   a) We develop a scalable metrics for success to determine the effectiveness of our work.

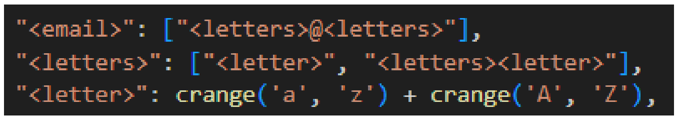
A. Fuzzingbook

The Fuzzingbook is a textbook with an accompanying python library created by Andreas Zeller. The textbook was designed for teaching the foundations of automating software testing. Because of its focus on educating and in-depth documentation, we thought it would be a great tool to extend.

Our project specifically extends the work of Chapter 5: Testing Graphical User Interfaces, but relies on the content and classes built throughout the book.

Because our project extends the Fuzzingbook capabilities, we work closely with its fuzzing pipeline. The Fuzzingbook interacts with a website through the Selenium Webdriver with the website running in a headless browser. This allows the tool to read the HTML content and interact with a website as a user.

The fuzzing process begins by identifying all interactable elements, or mining. Only elements that have been explicitly defined such as the *a* tag, *button* tag, and the *input* tag are string-matched in the HTML. These elements are known for being interactable. The set of unique interactable elements on a page for a state. Throughout the fuzzing process, new states may be combined if they have the same set of elements, which the tool handles. After determining the possible actions, each action is fuzzed. Since the possible actions types are known before hand, a constant grammar template is used to define possible fuzzing expansions.



```
"<email>": ["<letters>@<letters>"],
"<letters>": ["<letter>", "<letters><letter>"],
"<letter>": crange('a', 'z') + crange('A', 'Z'),
```

Fig. 1: Email Nonterminal Grammar Template

Fig 1 is a snippet from the grammar template defining the email attribute of an input element. A fuzzer will generate expansions knowing an email is made up of letters followd by an @ followed by more letters, and letters are a single letter or multiple. Prepossessing expansions allows the Fuzzingbook to quickly produce possible inputs. The tool then uses the Selenium webdriver to interact with the webpage. An interaction may lead to a new state and the process continues.

B. Webscraping

We begin with extending support for clearly interactive elements. The tool, as described previously, supports 3 tags with only few attributes of the input tag.

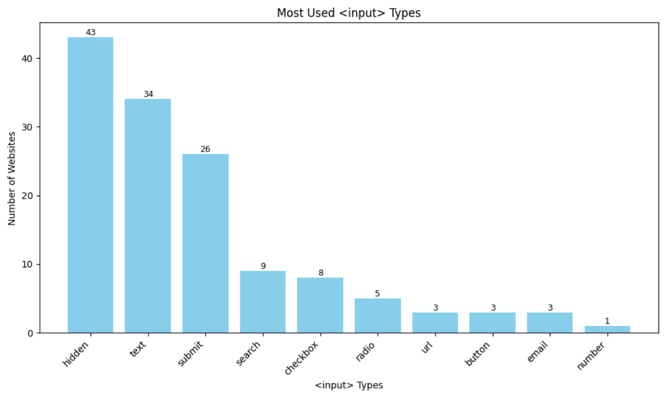


Fig. 2: Input Attribute Frequency on Popular Website Home Pages

The input tag is a clearly interactable element that we plan to extend so we explore what the most prevalent input tags are. Fig 2 shows the input attribute frequency of scraping the home pages of the top 100 most popular sites. Explore demersdesigns site list for further details. The Fuzzingbook tool actually supported many of these attributes, we added the search, url, and some less common ones such as color and range. The hidden attribute was the most frequent, but it is not something the user directly interacts with, as it is made to transmit information such as IDs or security tokens, so we did not fuzz it.

C. Test Setup

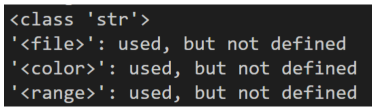
We test on a variety of environments to better understand the limitations of the Fuzzingbook and our additions. We built out a test website in Django to test certain fuzzing behavior because we know the expected state graph. We also fuzzed the top 10 most popular websites. The Fuzzingbook process still applies. The websites run in a headless browser and interaction takes place through the Selenium Web Driver.

D. Feature Additions

All features added work towards expanding the fuzzing tool's capabilities. We focused on a variety of features instead of quantity of additions within a feature because the ultimate goal was to learn more about the GUI fuzzing space through this project.

1) Software Design Practices: We tried to work with the library in a extensible way. This means that we did not edit the library and our work can be attached to new installations of the Fuzzingbook. Instead, all changes to the library were made through subclasses to either override functionality, or use enable use of features not previously setup in other parts of the library. The library is large, with over 30,000 lines of code with about $1/3$ of the lines relevant to our project. This made additions and changes non-trivial.

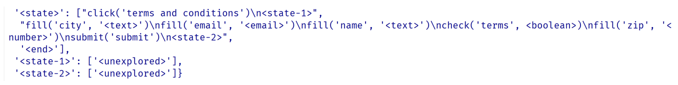
2) Crash Resistant Exploration: When we initially ran the tool, it would crash as shown in Fig 3.



```
<class 'str'>
'<file>': used, but not defined
'<color>': used, but not defined
'<range>': used, but not defined
```

Fig. 3: Log output before crashing

Further looking into the tool, this crash occurred because the grammar miner would ingest input tags with attributes like file, color, and range that it didn't support. Later in the pipeline, before actions are executed, the actions would be validated and since their was no grammar template for these attributes the tool would safe crash. The simplest fix would have been to remove ingestion of unsupported attributes. However, we liked this logging to gain a better understanding of the tool's limitations on modern sites.



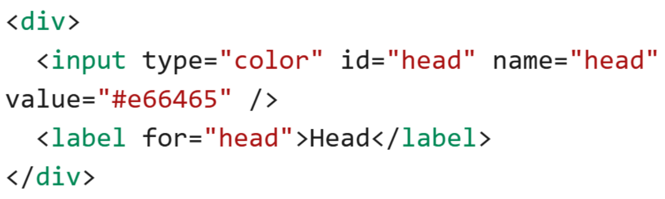
```
 '<state>': ["click('terms and conditions')\n<state-1>",
  "fill('city', '<text>')\nfill('email', '<email>')\nfill('name', '<text>')\ncheck('terms', <boolean>)\nfill('zip', '<
number>')\nsubmit('submit')\n<state-2>",
  '<end>'],
 '<state-1>': ['<unexplored>'],
 '<state-2>': ['<unexplored>']}
```

Fig. 4: State Grammar Example

Our solution was edit the state grammar as shown in Fig. 4. This is a dictionary of sorts that the fuzzingbook updates throughout exploration and uses for simple logging. Before actions are acted upon they are validated and stored here. Our solution removes every action that has an undefined nonterminal before execution of actions. For instance if the grammar didn't define expansion values for the ¡email¿ nonterminal then we would delete the fill action before all the actions are executed. This is a robust fix to allow easy logging with flexible element ingestion.

3) Input Type Extension: We identified important input types to expand and here we implement them. We go through the following process to add a input/attribute.

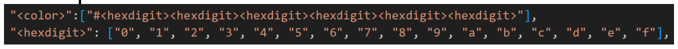
We first extend our subclass of the GUIGrammarMiner. This class handles finding interactable elements on an HTML page and creating actions that are added to the state grammar. Using the Selenium WebDriver, a string-matching mechanism is used to determine elements. To add the element in Fig. 5, we would search for the input tag with a attribute type of color.



```
<div>
  <input type="color" id="head" name="head"
value="#e66465" />
  <label for="head">Head</label>
</div>
```

Fig. 5: Color Input Element

After an action is added to the state grammar. We need to ensure that we support fuzzing an action with this attribute type. Fig. 6 allows Fuzzingbook's built in Fuzzer to efficiently generate possible values.



```
"<color>":["#<hexdigit><hexdigit><hexdigit><hexdigit><hexdigit><hexdigit>"],
"<hexdigit>": ["0", "1", "2", "3", "4", "5", "6", "7", "8", "9", "a", "b", "c", "d", "e", "f"],
```

Fig. 6: Color Grammar

Next we to add a mechanism to process the action. We use the Selenium WebDriver to interact with the webpage. Fuzzed values may create a new state and the process will continue.

We added the color, date, search and file input attributes. These additions followed this process with slight variations in how the actions were executed. We decided not to focus on further syntax extensions and instead focus on methods that would provide smarter fuzzing and more fluid element ingestion.

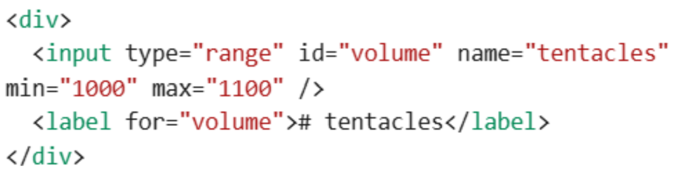
4) Enabling Advanced GUI Fuzzing: We wanted to pivot towards smarter GUI Fuzzing so before we added additions, we explored the library. We noticed that the GUICoverageFuzzer that our subclass relied on inherits from the GrammarCoverageFuzzer this class provides some efficient fuzzing at it ensures fuzz values are not repeated. However, for more powerful fuzzing, we wanted the GUICoverageFUzzer to inherit from a subclass of the GrammarCoverageFuzzer called ProbabilisticGeneratorGrammarCoverageFuzzer. This subclass is the combines all fuzzing features in the fuzzingbook enabling pre and post-filtering of grammar expansion and probabilistic grammar expansion, prioritizing one group of values over another.

We tried to cleanly change dependencies of the library, but ultimately settled with copying all the functionality of the GUICoverageFuzzer (ie. copying methods and class fields) and creating a subclass inheriting from this more powerful fuzzer. Original dependencies can be seen in Fig. 15 and Fig. 14. Both classes inherited from the same parent class.

We ultimately did not use the more powerful fuzzer to the fullest potential as we could not figure out how to combine our work with smart fuzzing with the fuzzingbook's pipline. However, enabling this feature was part of the exploration process and helped our learning of GUI testing strategies.

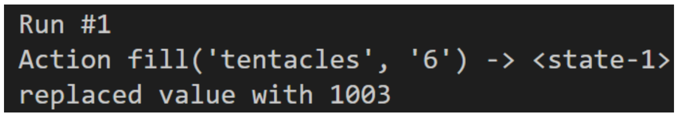
5) Fuzzing with constraints: Our next addition extended the tools ability to fuzz with more context. Specifically explicit constraint attributes. With onboarding of the more powerful fuzzer described in the previous section, we tried using the pre and post conditions for constraint handling but couldn't think of a clean way to integrate our work. Instead, we hook a new fuzzer before execution and refuzz a valid value if necessary.

In this example Fig. 7, the range component expects a bounded value. The fuzzer uses the grammar as outlined in Fig. 4 but does not consider these constraints. Thus, its value will most likely be out of range. Our re-fuzzed value considers the min and max values if present and generates a new input. By not using the fuzzing pipeline and keeping this input generation simple we lose out on the ability to pre-generate all possible inputs and prevent same input generation, but these additions are ultimately proof-of-concepts. For this feature we only on board the min and max constraints for number data types but could add date types in the future.



```
<div>
  <input type="range" id="volume" name="tentacles"
min="1000" max="1100" />
  <label for="volume"># tentacles</label>
</div>
```

Fig. 7: Range Component



```
Run #1
Action fill('tentacles', '6') -> <state-1>
replaced value with 1003
```

Fig. 8: Constraint Addition

6) Fuzzing with LLM: A major addition to the tool was the integration of LLM Fuzzing. This section discusses the process in implementing this feature and what we learned along the way.

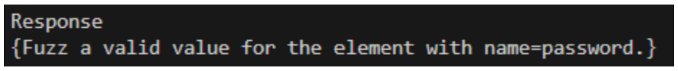
a) Integration Considerations: We used a variety of methods to integrate an LLM into our system. The system can be configured to support any LLM. In our experimentation, we considered compute, cost, and ease of setup. A large amount of effort was spent trying to configure the latest-and-greatest Llama 4 models, however, at the time when we started work on this stage of our work the models had only been released for 2 days. So as such, we encountered many source code bugs, specifically with the transformers library, when attempting to load the model onto the Rivanna HPC cluster. We then decided to pivot to llama-3.2-3B, which was a breeze to setup and run inferences on. However, we failed to integrate our local LLM inference into our fuzzing code work on the Rivanna cluster. Thus, we pivoted away from Rivanna and used free API inference endpoints on our local machines.

We used 3 API integrations that assisted with the same purpose, to query an LLM for free. First, we used a unofficial hugging chat API [2]. This essentially turned a chat window into the api allowing us to query an LLM for free. We eventually had rate limiting problems with this setup. Next we used the official hugging chat API until we ran out of our 10 cents of inference. We turned to other providers [1] like Together AI that gave us 1 dollar of inference which was enough to conduct our experiments. We learned that LLMs are not cheap and using them for something like functional testing will get expensive.

b) Input Considerations: We wanted to balance quantity with quality of context. This is because too much context to an LLM is expensive, but can also confuse smaller models, as it did in our test with Meta-Llama-3-8B-Instruct-Lite, when we gave it an entire HTML page.

We gave the model the following system prompt. We wanted instructed the model to put the value in curly brackets because we wanted an easily parse able way to retrieve the value. Unfortunately, smaller models like the Llama-8b wouldn't follow instructions as shown in Fig. 9 so the subsequent instructions seem to help a little. Using more powerful models like the Llama-4-Scout-17B-16E-Instruct returned results in their expected form. Through this testing with API's we could see that our Rivanna setup was limited as it could only support simpler models.

```
“Respond only with the input value
in curly brackets (eg. {value}) all
descriptors outside the brackets.
No descriptors should exist. Use
the HTML provided to create a valid
fuzz value for the element.”
```



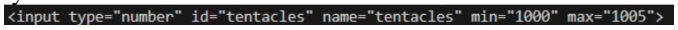
```
Response
{Fuzz a valid value for the element with name=password.}
```

Fig. 9: Invalid Response Output

We tried two methods for the actual prompt.

```
“Fuzz a valid value of the element
with name={name}. Use HTML context
if needed. Element-specific HTML:
{element_html} HTML context:
{html_context}”
```

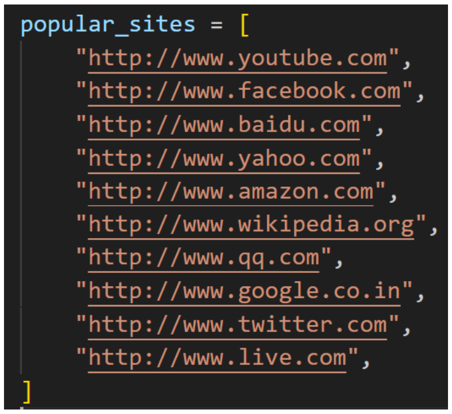
The first method we gave the LLM both the element html in Fig. 10 and the HTML context. The element HTML captures all context of the tag so was a great source for the LLM. The HTML context captured the larger context as a whole. In our case we set it to capture 2 parent levels from the current element, this ensures it doesn't return the whole page which would be too much context and too expensive while probably returning relevant info. Because we are using an API and have limited inference, we performed our in-the-wild testing with only the element HTML.



```
<input type="number" id="tentacles" name="tentacles" min="1000" max="1005">
```

Fig. 10: Element HTML

IV. Results



```
popular_sites = [
    "http://www.youtube.com",
    "http://www.facebook.com",
    "http://www.baidu.com",
    "http://www.yahoo.com",
    "http://www.amazon.com",
    "http://www.wikipedia.org",
    "http://www.qq.com",
    "http://www.google.co.in",
    "http://www.twitter.com",
    "http://www.live.com",
]
```

Fig. 11: Top 10 Popular Sites

3 versions of the tool were run on popular sites in Fig. 11. The base tool modified to prevent crashing was run. The tool with our input and constraint fuzzing additions was run, and the tool with previous additions plus LLM fuzzing was run. All logs and state space graphs are in the repository linked at the top of the page for further analysis. The results aim to quantitatively analyze this experiment through the graph properties. The tools were run for 20 actions, although they probably could have processed many more.

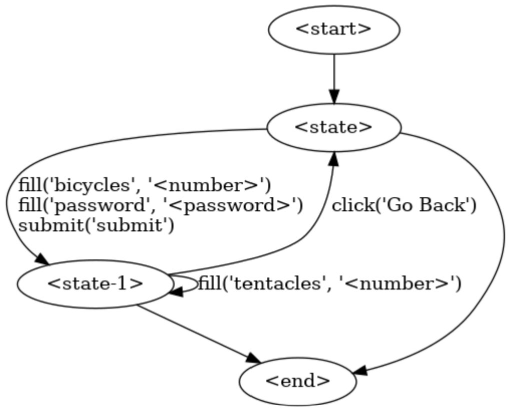

<start>
<state>
fill('bicycles', '<number>')
fill('password', '<password>')
submit('submit')
click('Go Back')
<state-1>
fill('tentacles', '<number>')
<end>

Fig. 12: Example Custom Site State Component Testing

A. State Graph Analysis

In the following tables, we report the highest centralitys of states that are not the ¡state¿, ¡start¿, ¡end¿, and ¡unexplored¿ states. Centrality is an interesting metric to apply to these state graphs since, intuitively, centrality is a measure of the most important nodes in a graph. The tables will report three different centrality metrics for each site collected for that experiment, where the cells will show which state had the highest metric followed by the score it received.

Website	Betweeness	Closeness	Pagerank
baidu	11: 0.0322	7: 0.0500	11: 0.0335
facebook	1: 0.0022	70: 0.0056	149: 0.0023
live	32: 0.0076	64: 0.0122	1: 0.0044
qq	1: 0.0500	1: 0.2667	1: 0.1384
wikipedia	1: 0.1667	1: 0.3333	1: 0.1998
youtube	1: 0.1667	1: 0.3333	1: 0.1819

TABLE I: Base Fuzzing Book GUI Fuzzer Centralities

Website	Betweeness	Closeness	Pagerank
baidu	3: 0.0323	15: 0.0484	1: 0.0184
facebook	24: 0.0081	25: 0.0122	25: 0.0049
live	1: 0.0000	1: 0.0202	1: 0.0082
qq	1: 0.0500	1: 0.2667	1: 0.1384
wikipedia	1: 0.1667	1: 0.3333	1: 0.1998
youtube	1: 0.1667	1: 0.3333	1: 0.1819

TABLE II: Extended Smart GUI Fuzzer Centralities

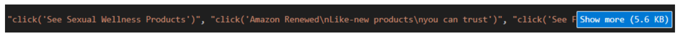
Website	Betweeness	Closeness	Pagerank
baidu	13: 0.0322	9: 0.0500	13: 0.0201
facebook	16: 0.0017	96: 0.0049	6: 0.0020
live	1: 0.0000	1: 0.0202	1: 0.0082
qq	1: 0.0500	1: 0.2667	1: 0.1384
wikipedia	1: 0.1667	1: 0.3333	1: 0.1998
youtube	1: 0.1667	1: 0.3333	1: 0.1819

TABLE III: Extended GUI Fuzzer Centralities

V. Discussion

A. *In-the-wild testing*

The detailed run logs and generated state space graphs can be reviewed in the repository linked at the top of this document. The state graphs generated are too large to easily display in this report. Testing our tool on the top 10 most popular websites yielded some valuable insights about fuzzing modern web frameworks. The first observation is that the interactable elements on a site can be quite large. Fig. 13 shows an example state space, where the number of actions logged is enormous. For the testing we had AI handle most input actions. With action pools as large as this, it is not feasible to fuzz all elements with AI as compute costs would outweigh its utility. The size of the state graph also slowed down the runtime as the activity was tracked in one big string variable.



```
"click('See Sexual Wellness Products')", "click('Amazon Renewed\nLike-new products\nyou can trust')", "click('See F
```

Fig. 13: State Space of amazon.com

The second observation across all tests was that the runs were non-deterministic, in the case of custom site testing, when the tool was only working with a few elements the results seemed pretty deterministic, but in the case where there were so many actions to choose from and the tool probably wouldn't process all actions before reaches the 20 action limit, the results varied per run. There were also some unexpected crashes with errors like unterminated string literal and invalid-selector-exception. These failures were definitely on our end but didn't show in custom site testing. As shown in Fig. 13, I believe the string literal error was caused from either the size of the actions or the unfiltered characters that make up the action names as the action names follow a strict form.

The third observation across all tests was that the exploration wasn't too deep. More levels in the state space would show the tool discovering more states, but this was definitely a drawback of our setup since we were using a coverage fuzzer that first tries to explore all states at a given level and we limited the actions to 20 when there could have been 100+ actions. We did however do a step-by-step exploration with the tool in a Jupyter notebook by manually choosing actions from the mined set and could reach some depth in the sites.

B. *State Graph Analysis*

Based on the tables, we can see that there was some change in node centralities, which implies some fundamental change in what states were explored. However, we observed no significant change in node centralities with the incorporation of smart fuzzing. This could partly be attributed to newly discovered states only being incorporated further down the tree. We also notice that for the most part, the most important nodes in the graph are usually further up the tree. This is in line with what we would expect, since a homepage will be an entry point to many other pages of the web application. A last thing of note is that the Baidu and Facebook sites had the most amount of variability between our three experiments, which could indicate that our extensions were able to make more gains for these sites because of some property of our approach. For example, perhaps the input attributes we added support for were more present for these sites. Ultimately though, our incorporation of smart fuzzing was unable to make meaningful differentiations due to our straightforward approach of asking for one-shot fuzzing inputs. We believe that had we had the LLM receive the results of its input and then select another one based on the prior attempt, we could have seen a more in-depth state exploration.

C. *Custom Site Testing*

To validate input extensions and test limits such as with more complex forms, we did some initial custom site testing. The site is a simple Django application a part of the repository. We created example components for each input that would link to another page with more components. In Fig. 12, we test a the tools ability to submit a form, which would send it to another page with more components. Custom site testing was important in the build process as we knew the expected graph. However, the fuzzing tool had problems when there was only one element on a page.

D. *Future Work*

We only used LLMs to generate smart inputs. However, LLMs could be used throughout the fuzzing pipeline including in identifying elements that are not explicitly interactable from the HTML/JS context.

E. Limitations

LLM fuzzing has major benefits over previous methods. Grammar templates don't have to be explicitly defined as an LLM can infer the output format from the given context. Constraints don't have to be explicitly defined for the same reasons. LLMs can keep chat history context for multi-input fuzzing which requires related inputs.

However, LLMs do have drawbacks, with a major one being compute. We have to consider how this tool will be used in industry. Most functional testing tools run every time there are major changes to an app. When an app is released or pushed to production, it must first pass Q/A tests. With Selenium, luckily these tests can be recorded and played back. That means at a minimum, the LLM needs to be used to generate test cases every time the test suite needs to be updated from site changes that break previous tests. Thus, businesses that use this tool will need to consider the tradeoff between compute costs and test suite generation.

VI. Conclusion

In this project, we demonstrated an extension of the Fuzzbook's GUI Ripper tool to support modern web frameworks. Due to time constraints, we did not wish to support all interactable widgets from Bootstrap, so instead we webscraped the 100 most popular websites to determine which input attributes to support. From there, we extended the grammars to handle these common attributes while also incorporating a more graceful error-handling pattern for future unrecognized grammars. We also incorporated an LLM to help guide a context-aware fuzzing input, and measured the resulting state graphs to determine if there were any significant changes in the centrality of the states/nodes. Ultimately, we found this project super insightful for understanding modern software development principles and GUI fuzzing techniques.

References

[1] *Inference Providers*. URL: https://huggingface.co/docs/inference-providers/index (visited on 04/24/2025).

[2] *Soulter/hugging-chat-api: HuggingChat Python API*. URL: https://github.com/Soulter/hugging-chat-api (visited on 04/24/2025).

[3] Hao Wen et al. *DroidBot-GPT: GPT-powered UI Automation for Android*. 2024. arXiv: 2304.07061 [cs.SE]. URL: https://arxiv.org/abs/2304.07061.

[4] Andreas Zeller et al. "Testing Graphical User Interfaces". In: *The Fuzzing Book*. Retrieved 2024-01-31 17:32:50+01:00. CISPA Helmholtz Center for Information Security, 2024. URL: https://www.fuzzingbook.org/html/GUIFuzzer.html.

VII. Appendix

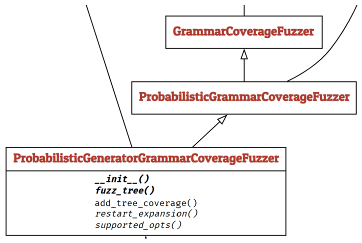


Fig. 14: PGGC Dependency on GrammarCoverageFuzzer

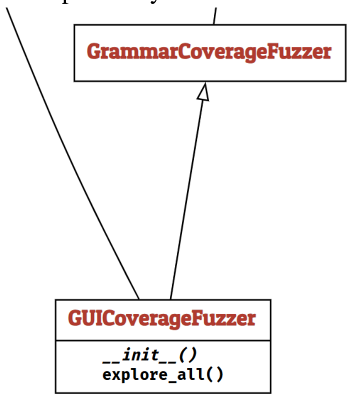


Fig. 15: GUICoverageFuzzer Dependency on GrammarCoverageFuzzer