Process Book

1. Introduction

Fuzz testing or fuzzing is an effective software testing technique that repeatedly generates test cases (i.e., software inputs) and feeds them to the software under testing to trigger bugs.

As the key component of fuzz testing, the test case generator constructs test cases that might reach potential bug-prone code using runtime feedback instead of blind generation. **Code coverage** is one such critical runtime feedback. With the collected runtime code coverage of each test case, fuzzing testing tools (also known as fuzzers) select test cases that hit coverage and further modify/mutate them for continuous testing. The intuitions behind the design are that (1) a test case hits new coverage is highly possible to trigger new coverage by slightly modifying it, and (2) maximizing the code coverage improves the possibility of finding bugs as no one knows where bugs are hidden. Although fuzzers can achieve higher code coverage by automatic mutations, they might fail in some complex functions or modules in programs under testing. **To know where the fuzzers are stuck and why the fuzzers cannot saturate these modules, it would be helpful to develop a visualized and hierarchical coverage plot that can offer a macro perspective while allowing detailed inspection at the module/function level.**

Further, bugs detected by fuzzers might be caused by the same root cause. Developers are burdened with an excessive manual effort to address the numerous **duplicated bug reports** fuzzers generate. Instead of manually analyzing these reports, **we propose clustering bugs semantically based on dimension-reduced embeddings to shed light on the characteristics and root causes of each bug cluster.**

Throughout this project, we achieved the following goals:

- Develop a tool, **FuzzVizz**, that can visually display hierarchical coverage and bug clusters.
- Apply FuzzVizz to real-world programs to show its flexibility.
- Conduct a preliminary evaluation with several fuzzing experts to demonstrate its practical utility.

2. Design

In our study, we chose compilers/interpreters as the programs under testing because their bug reports often include detailed root cause descriptions and proof-of-concept exploits. This helps us extract embedding from the text and create our visualization results.

We collected two types of data—coverage and existing bugs—to generate **coverage treemap** and cluster bugs from nine real-world programs, including Chakracore, CPython, Hermes, LuaJIT, MicroPython, MRuby, PHP, Ruby, and Webkit. Coverage data was obtained by running state-of-the-art fuzzers and analyzing the input corpus on a coverage-instrumented binary, while the existing bugs were collected by scraping bug issue trackers.

We detailed the processes of the two visualizations below.

2.1. Coverage Treemap

Visualizing coverage requires four steps: **target instrumentation, runtime tracing, aggregation, and coverage treemap generation**.

Target instrumentation adds additional code, that can record runtime coverage, to the program under test through compilers. The coverages are usually expressed as basic blocks or edges between basic blocks.

Runtime Tracing feeds inputs or test cases to the instrumented programs to collect coverage, which is often in a custom format (e.g., the Sancov format for Sancov instrumented programs). In our study, we adopt Sancov as our coverage format as it is widely used, collision-free, and convenient for visualization.

After collecting coverages, we need to **aggregate** them by computing the union of the covered basic blocks/edges.

Finally, we map this list back to the source code to **generate a human-readable coverage treemap**, as dumping code offsets in binary would not be too helpful for developers. Our coverage treemap organizes coverage into three hierarchies: folders, files, and functions, represented as tiles in the treemap. For example, Figure 1 shows

the treemap at the folder and file granularity, while Figure 2 displays at the function granularity.

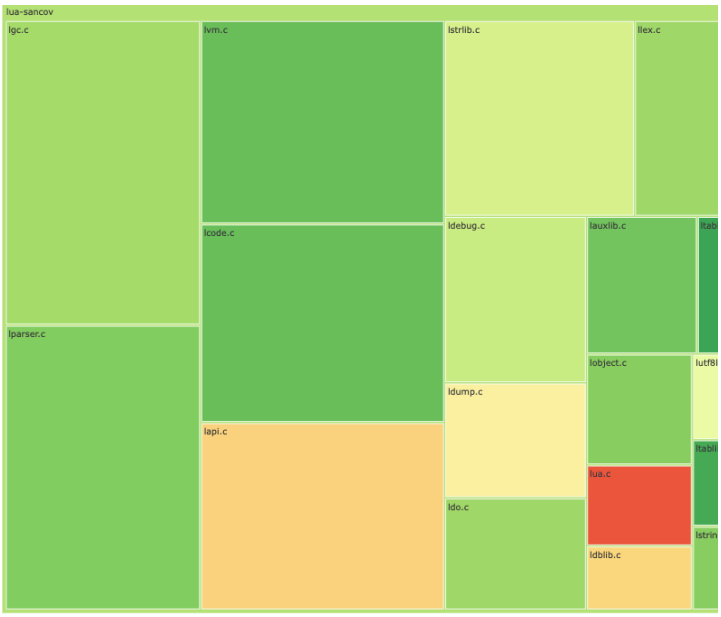

Figure 1. Treemap at the folder and file granularity

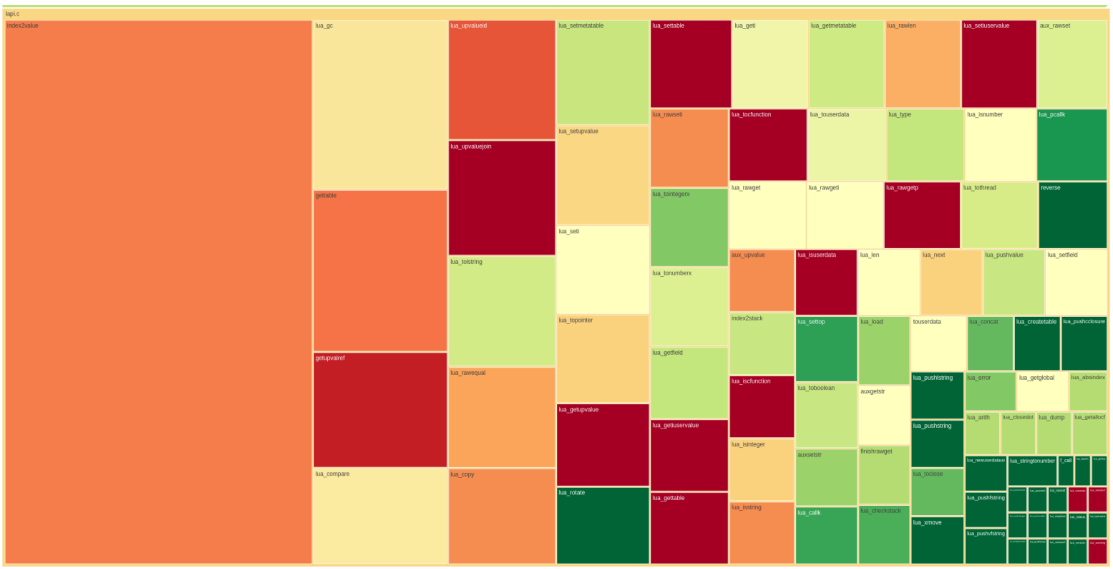

Figure 2. Treemap at the function granularity

The size of the tile indicates the number of control edges in the folder/file/function, and the color indicates the percentage of edges covered. Developers can focus on large and red blocks, representing significant chunks of uncovered code and potential fuzzer roadblocks. Users can navigate in and out of different rectangles within the treemap,

revealing more detailed function coverage and displaying coverage percentages and the total number of edges upon hovering.

2.1.1. Challenge

It is non-trivial to have such a visualization because of the performance challenge. For larger programs like V8, the traced coverage often contains hundreds of thousands of control edges per execution. Aggregating coverage for a typical fuzzing corpus containing tens of thousands of inputs was initially slow with a Python script. We improved performance by switching to the Polars library, which offers CSV parsing in Rust and SIMD-accelerated parallel data frame operations. This reduced the aggregation process to a few seconds for all programs.

2.1.2. Use case 1: Developer Creating a Test Suite

A developer inspecting the top-level treemap for Lua notices that the `lapi.c` block is large with a relatively low coverage percentage (38.19%). By zooming in, the developer sees individual function coverages, discovering that many functions related to "upvalues" are uncovered. This insight leads the developer to modify the fuzzer's mutator to create more inputs that involve variable capturing closures and manually crafting test cases to exercise this feature.

2.1.3. Use case 2: Comparing Two Fuzzers

Developers can also utilize FuzzVizz to compare coverage treemaps from different fuzzers to reveal the different coverage patterns.

For example, when comparing Nautilus (a grammar fuzzer) and AFL (a byte-level fuzzer) on mruby, as shown in Figure 3 and Figure 4, we can see that AFL++ performs better at stressing the parser but fails to exercise the semantic stage due to early interpreter exits on syntax errors. Focusing on syntactically correct inputs, Nautilus shows higher coverage in files like `gc.c` and `array.c`, exercising the semantic stage more thoroughly. This comparison demonstrates that AFL++ and Nautilus exercise different functionalities of mruby.

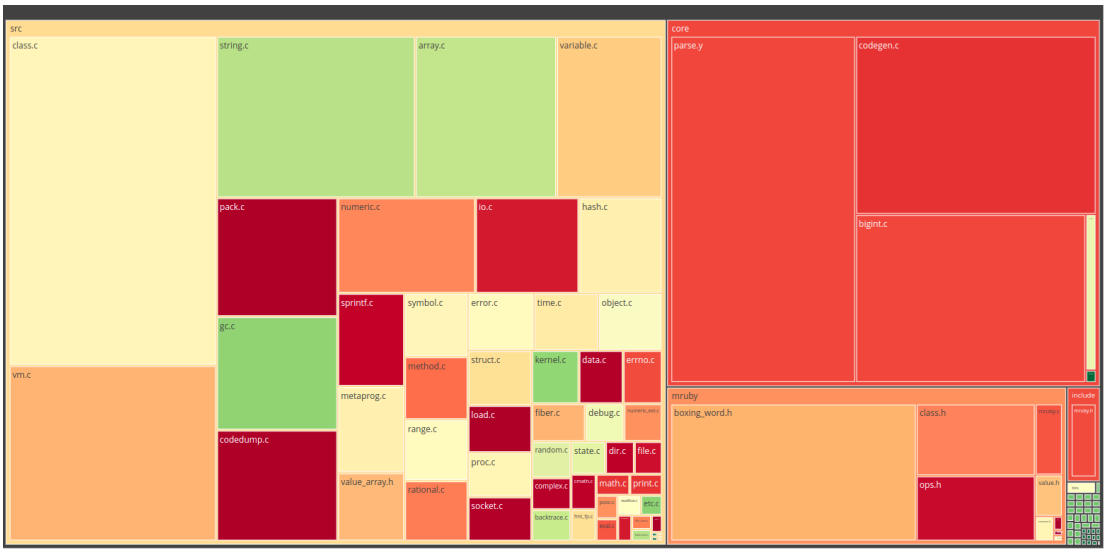

Figure 3. The coverage treemap of Nautilus

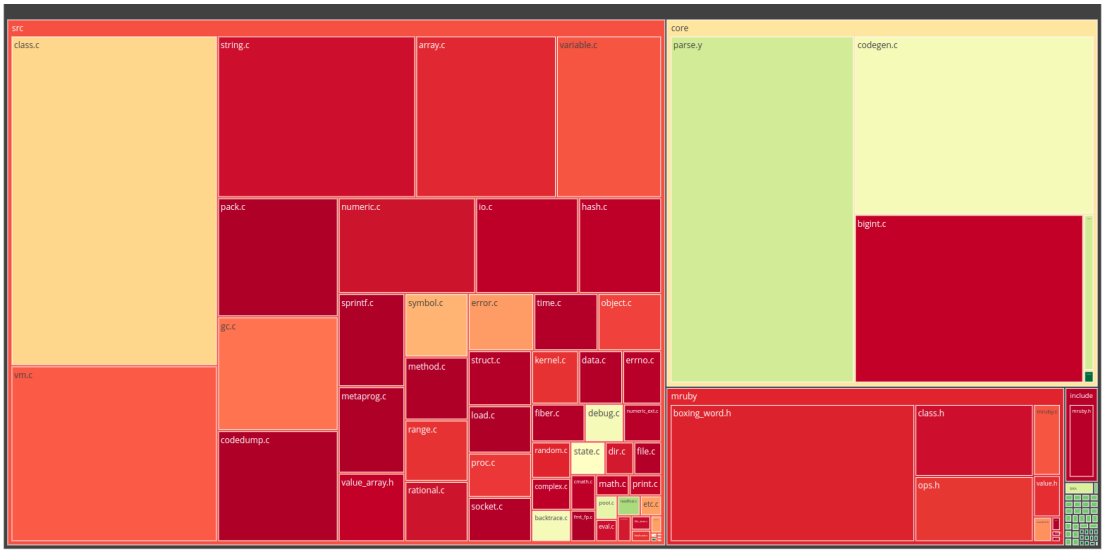

Figure 4. The coverage treemap of AFL++

2.2. Bug Clustering

We aim to identify bug patterns that may point to potentially vulnerable code areas requiring more fuzzing or refactoring. Searching bug issue trackers for specific keywords or labels can be laborious, and some bug reports may not contain specific keywords. We propose a bug clustering scatter plot (or a map of bug reports) where similar bugs are semantically mapped closer together, as Figure 5 shows. Users can hover over a dot to see a bug report summary, and topics are automatically assigned to each cluster to characterize a bug class.

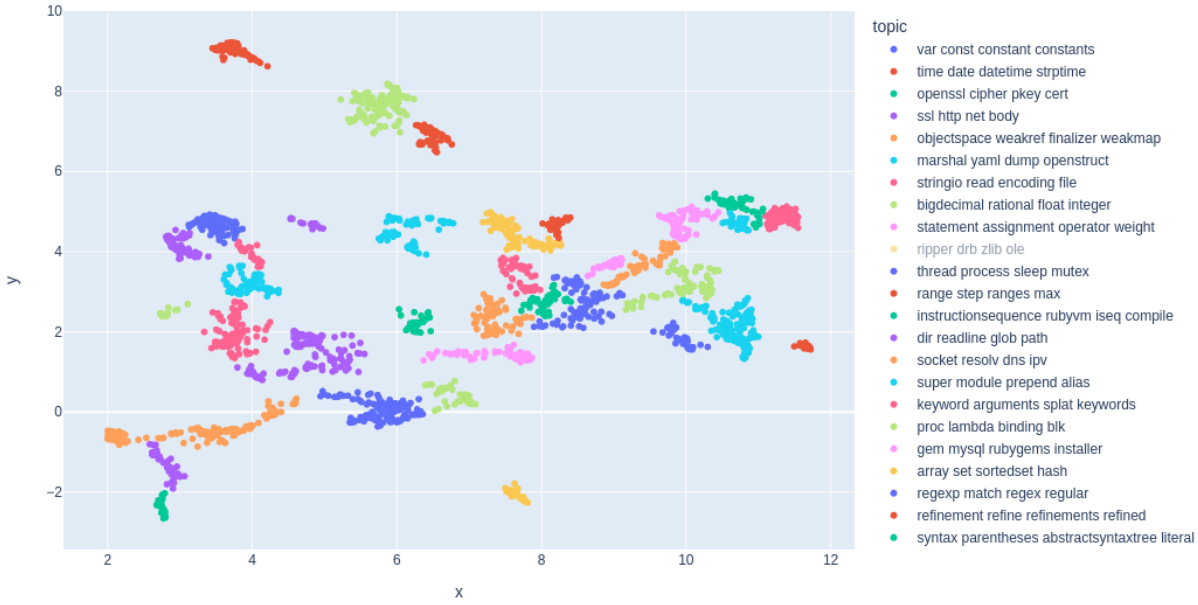

Figure 5. A bug clustering example

The above goal is accomplished with the following procedures:

- **Summarization:** Noisy information in bug reports can adversely influence clustering, so we prompt GPT-4 to summarize bug reports, reducing verbose stack trace dumps and lengthy discussions among reporters and developers.
- **Embedding Extraction:** We use the state-of-the-art text-embedding-3-large models from OpenAI to extract embeddings, outputting a 3072-dimension vector for each summarized bug report. This vector represents the "meaning" of the report.
- **Dimension Reduction:** High-dimensional data is hard to visualize, so we reduce it using UMAP, which offers better performance and clustering quality compared to PCA and t-SNE. UMAP quickly returns results, making it suitable for large datasets like CPython's 90,000+ bug reports.
- **Clustering:** We use HDBSCAN for clustering, which empirically provides the best cluster segmentation without requiring a predefined number of clusters.
- **Topic Assignment:** We calculate cTF-IDF (class-based Term Frequency - Inverse Document Frequency) for each token in the bug reports and assign the top-5 tokens as the cluster topic. This helps in understanding the common theme of each cluster.

- **Plotting:** We create a scatter plot where coordinates represent the dimension-reduced vector, colors represent different bug clusters, and topics provide a rough idea of each cluster.

2.2.1. Challenge

The main challenge was learning machine learning and natural language processing techniques. Most team members came from a systems programming background and were not initially familiar with concepts like embeddings, clustering, and topic assignment. Learning these new technologies was initially intimidating but ultimately rewarding and beneficial. Understanding that we can represent the "meaning" of a piece of text with embeddings and quantify the similarity between texts using vector distance was particularly exciting and useful.

2.2.2. Use case: Studying Bug Patterns

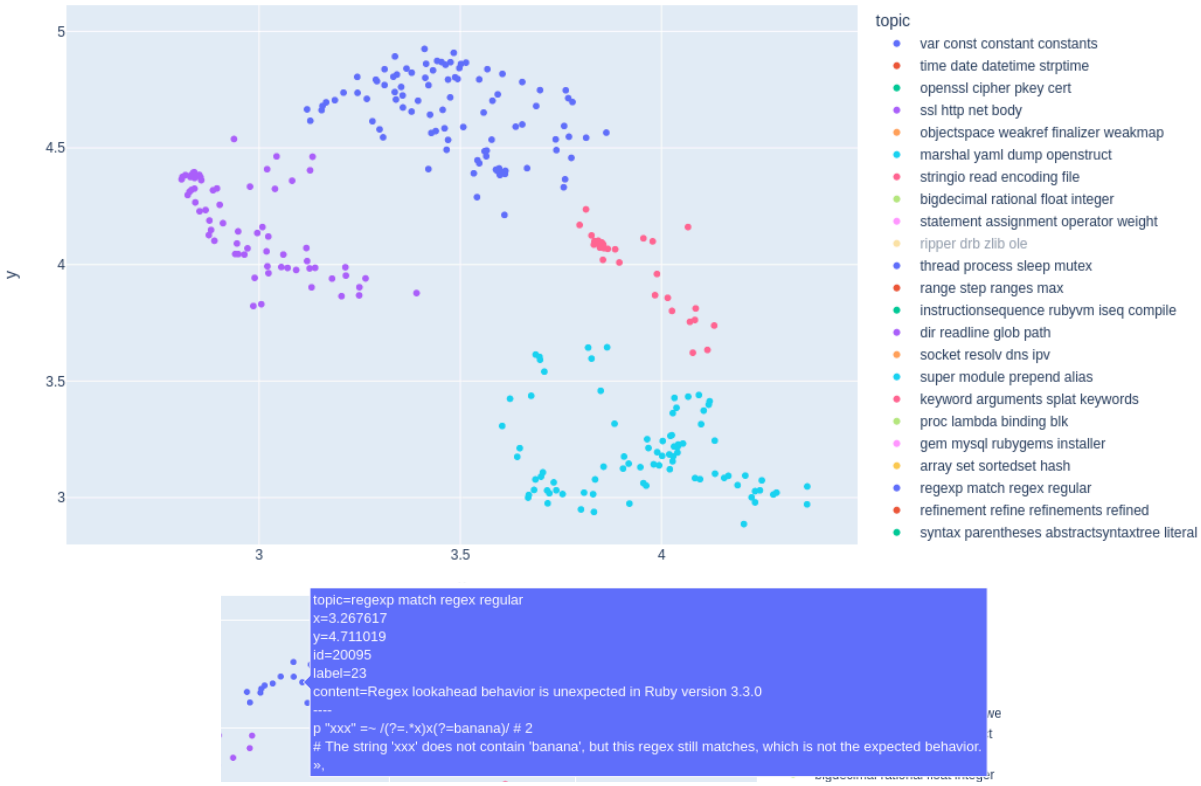

Figure 6. A bug pattern in the Ruby interpreter

By examining the bug clustering plot for the Ruby interpreter shown in Figure 6, we discover that certain clusters indicate string-related issues (e.g., Unicode, regular expressions, escape

characters, and encoding issues). This knowledge helps developers and auditors test these functionalities more thoroughly. For example, they might test string operations like concatenation, search, and regex matching in different encodings and conversion between different encodings.

3. Conclusion

In this project, we developed FuzzVizz, a tool for visualizing hierarchical code coverage and clustering bug reports. We applied FuzzVizz to real-world programs and demonstrated its flexibility and practical utility through preliminary evaluations with fuzzing experts. The coverage treemap and bug clustering plot provides valuable insights for developers to improve fuzzing efficiency and understand bug patterns. Despite the challenges in performance optimization and learning new technologies, the project successfully contributed to better software testing practices.

4. Peer Assessment

Our team includes Chibin Zhang, Zurab Tsinadze, and Tao Lyu. The breakdown of the parts of the project completed by each of us is listed below:

- Chibin Zhang
    - Created clustering pipeline.
    - Worked on the treemap plot and the cluster plot.
    - Drafted the process book.
- Zurab Tsinadze
    - Worked on the line of code growth chart and buildings visualization.
    - Created the website prototype and later finalized it.
- Tao Lyu
    - Orchestrated and engineered the fuzzing campaigns.
    - Extracted coverage information from the campaigns.
    - Finalized the process book.

Old

Data Collection

For our study, we chose compilers/interpreters as the programs under testing because their bug reports often include both root cause descriptions and proof-of-concept exploits, which helps us to fine-tune our visualization results. We scraped bug issue trackers for nine real-world programs: Chakracore, CPython, Hermes, LuaJIT, MicroPython, MRuby, PHP, Ruby, and Webkit. Coverage data was obtained by running state-of-the-art fuzzers and analyzing the input corpus on a coverage-instrumented binary.

Coverage Treemap

Visualizing coverage often requires four steps: *Target instrumentation, Runtime Tracing, Aggregation, and Report generation.* Programs with coverage instrumentation logs which basic blocks / control flow edges are reached during execution. Usually when obtaining coverage, a testsuite of inputs is run on the instrumented program. And as a result of execution, the executable will dump a list of covered blocks and edges, often in custom format, for example `sancov` instrumented programs will dump a list of covered basic block local (program counter offset in the `.text` section) in its custom `.sancov` format. Since our goal is to visualize coverage for fuzzers, we choose `sancov` as it is widely used by fuzzers and also collision-free. Because each run will produce a different trace, we need to aggregate it together, typically this means computing the union of the covered basic blocks / and edges. Now we have a list of covered code locations, we still need to mapped it back to source code and generate a human-readable report since dumping code offset in binary wouldn't be too helpful for the developer.

Our coverage treemap organizes coverage into three hierarchies: folders, files, and functions, represented as tiles in the treemap. The size of the treemap indicates how many control edges are there in the folder/file/function. The color indicates the percentage of edges covered. In general developers want to focus on **large** and **red** blocks because they represent a significant chunk of uncovered code and potential fuzzer roadblocks. The depicted figure only shows two levels of hierarchies: folders and files. But if a developer would like more details about specific function coverage, Users have the ability to navigate in and out of different rectangles within the treemap. For instance, clicking on a rectangle tagged with a file name will reveal different function coverages within that file. Additionally, hovering over each rectangle will display the coverage percentage along with the total number of edges.

User story 1: developer creating a testsuite. Here we showcase a user story where our coverage treemap can be used to discover coverage roadblocks in fuzz targets. Upon inspecting the top-level treemap for `lua`, the developer noticed `lapi.c` is a large block with relatively low covered percentage. He hovers the mouse over the `lapi.c` block and sees that only 38.19% of the edges in this block is covered. Curious about why coverage is so low in this file, he clicks on the tile and and treemap zooms in, showing individual function coverage (in the lower figure). The inner tile shows quite a few functions that are not covered at all (dark red tiles). These uncovered functions share a common part in their name `upvalue`, and it turns out upvalues are related to variable captures and closures in lua. And this feature is under-exercised in the fuzzer generated corpus. With this knowledge in mind The fuzzer developer may choose to modify the fuzzer's mutator to create more inputs that create variable capturing closures and manually craft test cases that exercise the closure feature.

User story 2: comparing two fuzzers. The above two figures show the coverage treemap for `mruby` by replaying the corpus of Nautilus (a grammar fuzzer, above) and AFL++ (byte-level fuzzer, below). Both fuzzers achieve around 30% of the total edge coverage. Using this single metric it might be tempting to conclude the two fuzzers perform similarly, but the coverage treemap tells whole another story. AFL++ is a general purpose fuzzer that performs bit and byte level mutations. This kind of mutation is good for stressing the parser, as depicted in the figure above, but fails to exercise the semantic stage of mruby, since the interpreter exits early when encountering syntax errors. Nautilus, on the other hand, is a grammar fuzzer that focuses on creating syntactically correct inputs. When compared to AFL++, It has a very low coverage in `parse.y`, but much higher coverage in other files such as `gc.c`, `array.c`, exercising the semantic stage of `mruby` more extensively. From the treemap, we can thus conclude that AFL++ and Nautilus exercise **different** functionalities of `mruby`. As a further step, we plan to create a *diff* variant of the treemap where we subtract the edge coverage of one fuzzer from another, and use it as the color. This allows for more straightforward comparison of two fuzzers in a single figure.

Challenge. The main challenge we encountered during was the performance of the implementation. For larger programs like v8, the traced coverage often contains hundreds of thousands of control edges. Note that this is for a single execution. A typical fuzzing corpus contains tens of thousand input programs. Our initial aggregation script is written in python, parses the traced coverage from CSV format, and iterate through entries one-by-one. This worked fine for smaller programs like `mruby` but was problematic for larger programs like v8, which took around half an hour. We later discovered a performant dataframe library called `polars` which provides CSV parsing in Rust, and SIMD accelerated parallel dataframe operations (like `groupby()` and `uniq()`). We modified our implementation to polars, and now the coverage aggregation process takes no more than a few seconds for all programs.

Bug Clustering

The motivation for bug clustering comes as follows: we would like to look at similar bugs and deduce a bug pattern. The bug pattern could point to a potential vulnerable code area that deserves more fuzzing or refactoring. While most open-source software have a bug issue tracker such as GitHub issues, or BugZilla, searching for keywords / specific bug labels can be a laborious process. Moreover, some bug reports may not directly contain specific keywords but instead phrased in a different way. And labels may be incomplete. Therefore, we propose the bug clustering scatter plot (or a map of bug reports). Bugs reports are projected as dots in this map, and semantically similar bugs are mapped closer together. The user can hover the mouse over a dot to see a summary of the bug report. We also automatically assign topics to each bug cluster to characterize a bug class. In the following, we describe the process of turning a corpus of bug reports (text) into the sample figure, along with challenges and design choices we made during the process.

The process can be divided into six steps:

1. **summarization.** The crawled bug reports may contain text that is not directly related to the root cause of the bug such as verbose stack trace dump and lengthy discussion among the reporter and developers. This noisy information will adversely influence later steps such as clustering, topic assignment, as well as the visualization (we can't show the entire bug report, it's just too long). Therefore we prompt GPT-4 together with the bug report for a summarization. The prompt template is already shown in milestone 1.

2. **embedding extraction.** To group similar bug reports together, we need to define what's the "meaning" of a report, and a "similarity" metric. A common approach used here is to extract the embedding of a piece of text. We again leverage state-of-the-art `text-embedding-3-large` embedding models from OpenAI. The output is a 3073 dimension vector that corresponds to the "meaning" of the summarized bug report.
3. **dimension reduction.** Note that high dimensional data is very hard to visualize on a computer screen (which is only two dimensional). After a bit of research, we narrowed down to three dimension reduction techniques: PCA, t-SNE, and UMAP. PCA offers explainability, as the reduced vector is always a linear combination of the original feature vector. However, it is usually not very good for clustering when compared to t-SNE or UMAP. Our experimentation on t-SNE and UMAP gives similar clustering quality, but UMAP offers much better performance. For CPython which contains more than 90,000 bug reports, t-SNE hangs indefinitely while UMAP quickly returns results. Therefore, we settled on UMAP as the dimensionality reduction technique.
4. **clustering.** With the dimension reduced vectors (2d), we can already create a scatter plot but the problem is that all of the reports will belong to the same group, thus not very informative. Here we utilize clustering algorithms and assign group labels to each cluster. We tried three alternatives: K-means / Spectral clustering / and HDBSCAN. K-means and spectral clustering both require manually supplying the number of clusters K, while HDBSCAN require the minimum cluster size. We fiddled with the parameters a bit and empirically HDBSCAN gives the best cluster segmentation.
5. **topic assignment.** Note that the output of clustering algorithms are numeric labels. For better interpretability, we would like a textual description of what each cluster does. We tokenize each bug report and record the frequency of each token. We then calculate the cTF-IDF (class-based Term Frequency - Inverse Document Frequency) for each token. The top-5 alphanumeric tokens are sampled as the topic of the cluster.
6. **plotting.** With all the previous preparation done, we can finally create a scatter plot. The coordinate of each report is dimension reduced vector (2d). Different colors represent different groups of bugs (different root cause and component), and topics are assigned to give a rough idea of a bug cluster.

User story: studying bug patterns. In the following, we discuss how the bug clustering plot could be used to understand bug patterns. The above figure shows a map of bugs in the Ruby interpreter. When zooming into the highlighted region (red border), we do discover something interesting here: they are all string related issues! The clusters in purple, dark blue, pink, and light blue respectively indicate *unicode*, *regular expression*, *escape characters*, and *encoding* issues. Hovering over a dot gives more details about the bug report (shown below). With this knowledge in mind, a developer / auditor may choose to test these functionalities more thoroughly together. For example, trying to perform string operations like concatenation / search / regex matching in an encoding different from utf-8, as well as conversion between different conversions.

challenges: The main challenge we encountered in making the bug clustering plot is getting familiar with ML and NLP techniques. Team members mostly come from a systems programming background and were not very familiar with “embeddings” / “clustering” / “topic assignment”. Learning new concepts can be intimidating at first, but once we get used to the technologies it becomes very convenient and useful. Especially when we learned that we can represent the “meaning” of a piece of text with embeddings, and quantify how similar two texts are using vector distance! Was great fun learning this.

Peer Assessment

Chibin

- Worked on the treemap plot and the cluster plot
- Drafted the process book.

Zurab

- Worked on the line of code growth chart and buildings visualization.
- Created a website prototype.

Tao

- Orchestrated and engineered the fuzzing campaigns.
- Extracted coverage information from the campaigns.
- Together with Zurab worked to finalize the website.

Smooth team experience!