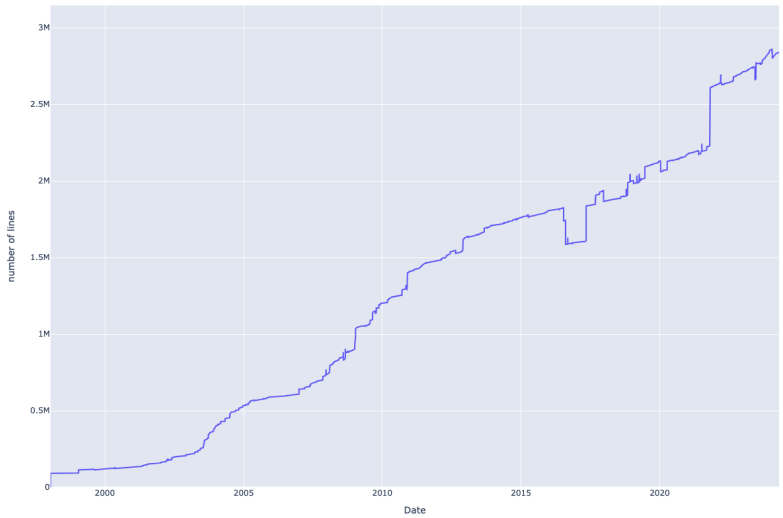
Mileston 2 - FuzzVizz

The aim of our project is to develop tooling and visualizations that facilitate two crucial aspects of software (fuzz) testing: hierarchical code coverage and semantic clustering of bug reports. Such visualizations help fuzzing experts and developers gain insights into the effectiveness of fuzzing campaigns and identify patterns within bug reports, ultimately enhancing debugging and duplicate elimination processes.

After consulting with security researchers and fuzzing experts, we have decided on the following visualizations:

1. Growth of lines of code of different projects over time. This shows the motivation of the problem, that modern software is incredibly complex, and searching for a bug in millions of lines of code is like finding a needle in a haystack.
2. Hierarchical code coverage after a fuzzing campaign that can offer a macro perspective while also allowing detailed inspection at the module/function level to see which regions of code were covered by testing. The same plot can be used to display the difference between two fuzzing campaigns as well.
3. Semantical clustering of bug reports based on dimension-reduced embeddings to shed light on the characteristics and root causes of each bug cluster.

Lines of Code

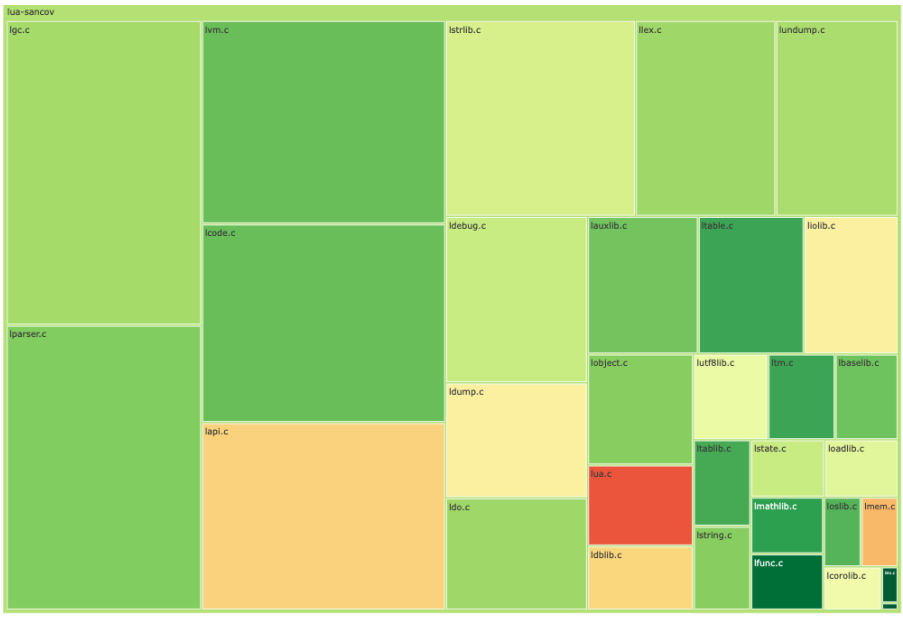


To retrieve the LoC of different projects over time, we will rely on Git history. Conveniently, each commit comes with metadata showing the number of lines that were added and removed. Starting from the first commit, we will iteratively build the dataset. Our visualization will be interactive, allowing users to zoom in on any date or time interval, regardless of its size. The total LoC after all commits will be plotted.

Hierarchical Coverage

Mileston 2 - FuzzVizz

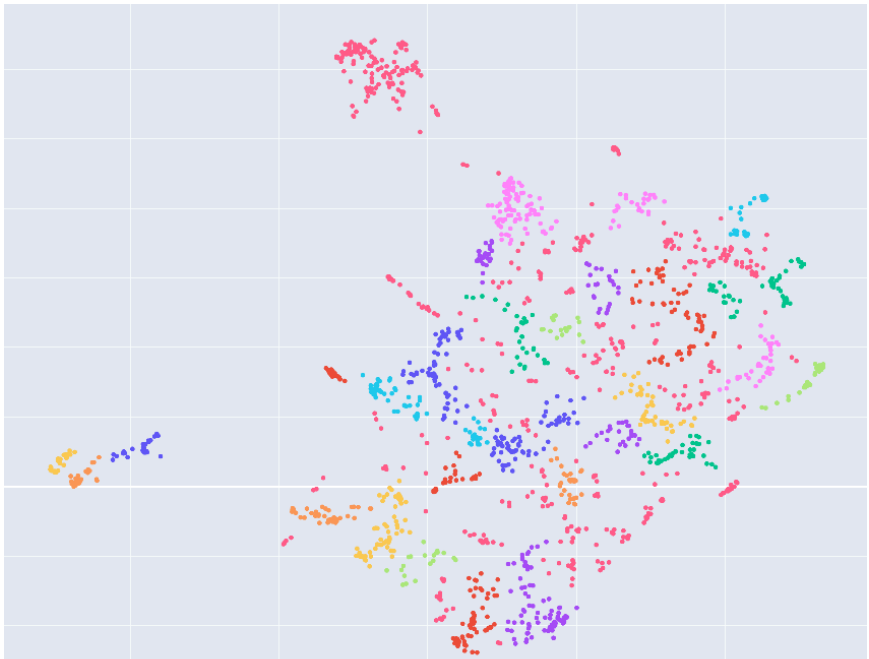
To gather the data, we will conduct fuzzing campaigns on target projects. This will yield a seed corpus, a set of inputs that achieves potential coverage within 24 hours. Subsequently, we will rerun coverage-instrumented binaries with this corpus to obtain final coverage across various granularities, including module, function, line, and edge levels. To showcase this data, we'll



utilize a recursive treemap visualization. Users will 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.

Bug Report Clustering

To obtain the data, we will scrape bug issue trackers from real-world programs, preprocess the data, and feed it into our clustering pipeline, similar to BERTopic. We will visualize the clusters



on a scatter plot with different colors. When users hover over a point, a tooltip will display the description of the report. Users will also be able to zoom in on any region of the plot and toggle different topics on or off.

Necessary tools to achieve our goals

Python and its libraries (scripts to fetch the data, pre-process, and cluster), GPT-4 (to extract useful information from bug reports), AFL++ (state-of-the-art fuzzer), plotly/d3 for visualizations, HTML/CSS/Javascript for the website.

Related Lectures

Lectures 1, 2, and 3 - HTML, CSS, and Javascript are needed for the website.
Lecture 4 - We will store our data in CSV and JSON formats.
Lecture 5 - Our visualizations will be interactive, give useful information on hover, and have the ability to zoom in and explore different regions.
Lecture 6 - Colors will differentiate bug report clusters, as well as show how well different regions are covered by testing (the greener the better).
Lecture 10 - We will use the TreeMap graph to visualize coverage data.
Lecture 12 - Storytelling is important to contextualize and motivate our visualizations.

Possible Enhancements

1. Create interactive time series visualizations that show the change in code coverage metrics (such as line, function, or branch coverage) over time for different projects. This can help users track the effectiveness of fuzzing campaigns and identify periods of rapid improvement or stagnation.
2. From Prof. Mathias Payer's Software Security lecture: The total lines of codes of Chromium and OS (together with relatively minor software that is needed to run the browser) is around 100 million. To better illustrate our idea that software is incredibly complex, imagine we print this code. Assuming 27 lines on a page and 0.1mm thickness it gets us a stack of 370 meters.
   We could add the visualization comparing 370m of paper to different high-rise buildings around the world.
3. Extend the bug report clustering visualization to include a temporal dimension, allowing users to see how bug report clusters evolve over time. This can help identify recurring issues or trends in software development and prioritize bug fixes accordingly.
4. Display hierarchical tree diagrams to compare the code coverage achieved by different fuzzing campaigns. This can provide a structured overview of how testing efforts have evolved over time and highlight areas of the codebase that have received more or less attention.