Google Summer of Code DeepMind

Streamline experiment execution and improve report UI for OSS-Fuzz-Gen

Expected Size: 350 hours

Mentor: Dongge Liu

Myan Vu
The University of Auckland
Auckland, New Zealand (UTC+12:00)
Github: https://github.com/myanvoos
Email: myanvoos@gmail.com | myan.vu@auckland.ac.nz
Personal website: https://myanv.dev
LinkedIn: https://www.linkedin.com/in/myanv/
Preferred method of contact: Email, Slack, Google Chat/Meet

Table of Contents

Abstract

OSS-Fuzz-Gen is a framework for fuzz target generation and evaluation by the Google Open-Source Security team. It generates fuzz targets for C/C+/Java/Python/Rust projects using LLMs and benchmarks them via the OSS-Fuzz platform. After running experiments in OSS-Fuzz-Gen, a report is generated and serves as the primary touchpoint for developers to understand their fuzzing results. As such, it's important that this interface is clear, intuitive, and provides meaningful insights into the fuzzing performance and coverage metrics.

This project proposes an updated UI to improve the results visualisation, report layout, and navigation, alongside feature additions such as a comprehensive search system, export, table of content, etc. It also proposes ways to streamline the CI pipeline which enables efficient experiment execution by introducing more automation processes.

Experiment Workflows (CI)

Currently the workflow is as follows. First a maintainer comments

```
/gcbrun exp -n <name> -m <model> -b <benchmark-set>
```

This triggers a Cloud Build trigger that runs `ci/ci_trial_build.py`:

```
def get_latest_gcbrun_command(comments):
    """Gets the last /gcbrun comment from comments."""
    for comment in reversed(comments):
        if body.startswith('/gcbrun exp'):
            args = parse_gcbrun_args(body)
            return args
```

The CI then uses these args to create a GKE job using `ci/k8s/pr-exp.yaml`:

```
containers:
- name: experiment
  image:
us-central1-docker.pkg.dev/oss-fuzz-base/testing/OSS-Fuzz-Gen-pull-request:
pr-${PR_ID}
  command: ["/bin/bash", "report/docker_run.sh", "${GKE_EXP_BENCHMARK}",
"${GKE_EXP_NAME}", "${GKE_EXP_FUZZING_TIMEOUT}", "ofg-pr",
"${GKE_EXP_LLM}"]
```

The experiment runs and results are uploaded to the GCS bucket:

```
$PYTHON -m report.trends_report.upload_summary \
  --results-dir ${LOCAL_RESULTS_DIR:?} \
  --output-path
"gs://oss-fuzz-gcb-experiment-run-logs/trend-reports/${GCS_TREND_REPORT_PAT
H:?}" \
  --name ${EXPERIMENT_NAME:?} \
  --date ${DATE:?} \
  --url "https://llm-exp.oss-fuzz.com/Result-reports/${GCS_REPORT_DIR:?}"
```

The results are then accessible via URLs.

This can be streamlined by having a CI process that continuously checks the status of the Docker image build process via polling and reports back in the form of GitHub comments. Once the experiment starts, automatically print links similar to here as a GitHub comment so that maintainers can monitor the ongoing experiment build process.

Report UI

I've built a prototype of the proposed report UI on a fork of the OSS-Fuzz-Gen repository. Since this is quite a complex repository, my aim was to avoid spending time on idealised, standalone mockups using placeholder data and instead work within the actual environment of OSS-Fuzz-Gen (with all its existing structures and considerations i.e. three-pages hierarchy, Jinja templates) from the outset, understanding firsthand what's feasible for implementation and how to actually go about implementing them.

For ease of viewing, the proposed changes to the report UI are classified into categories:

- [NAV]: Navigation
- [UI]: Readability & visual appeal
- [AGG] Aggregated metrics
- [VIZ]: Plots and visualisation
- [FEAT] Features

Setting The Stage

Dependencies

The prototype makes use of Alpine.js, chart.js , and Tailwind CSS. These lightweight dependencies can all be imported via CDN injection without requiring any complicated Node.js setup, fitting for both cloud-based and local environments. Since the trends report already uses `d3.js`, I will also migrate to using that instead of `chart.js` during the community bonding period to avoid bloating up the report's JavaScript.

Unified JSON

Currently, experiment data is split between the project-level `index.json` and the benchmark-level `crash.json`. While this makes sense given the fact that OSS-Fuzz-Gen needs to be compatible with large experiments, cloud logs, and CI artifacts, entirely separate JSON files cannot fully capture the hierarchical structure and relationships between projects, benchmarks, and samples. For this, we would require a unified JSON.

I want to be cautious here because trying to load the JSON with a large number of projects might lead to out-of-memory errors. Some optimisation techniques might be needed, such as loading the JSON once then performing search/filter operations in memory.

Proposed JSON structure:

```
{
  "project": "string",
  "benchmarks": {
      "[benchmark_name]": {
          "samples":
            [{
              "sample": "string",
              "status": "string",
               ...
            }],
          "build_success_rate": "number",
          "crash_rate": "number",
          "average_coverage": "number",
          "average_line_coverage_diff": "number",
          "total_coverage": "number",
          "total_line_coverage_diff": "number"
      }
  },
  "average_build_success_rate": "number",
  "average_crash_rate": "number",
  "average_coverage": "number",
  "average_line_coverage_diff": "number",
  "ofg_total_new_covered_lines": "number",
  "ofg_total_covered_lines": "number",
  "existing_total_covered_lines": "number",
  "existing_total_lines": "number",
}
```

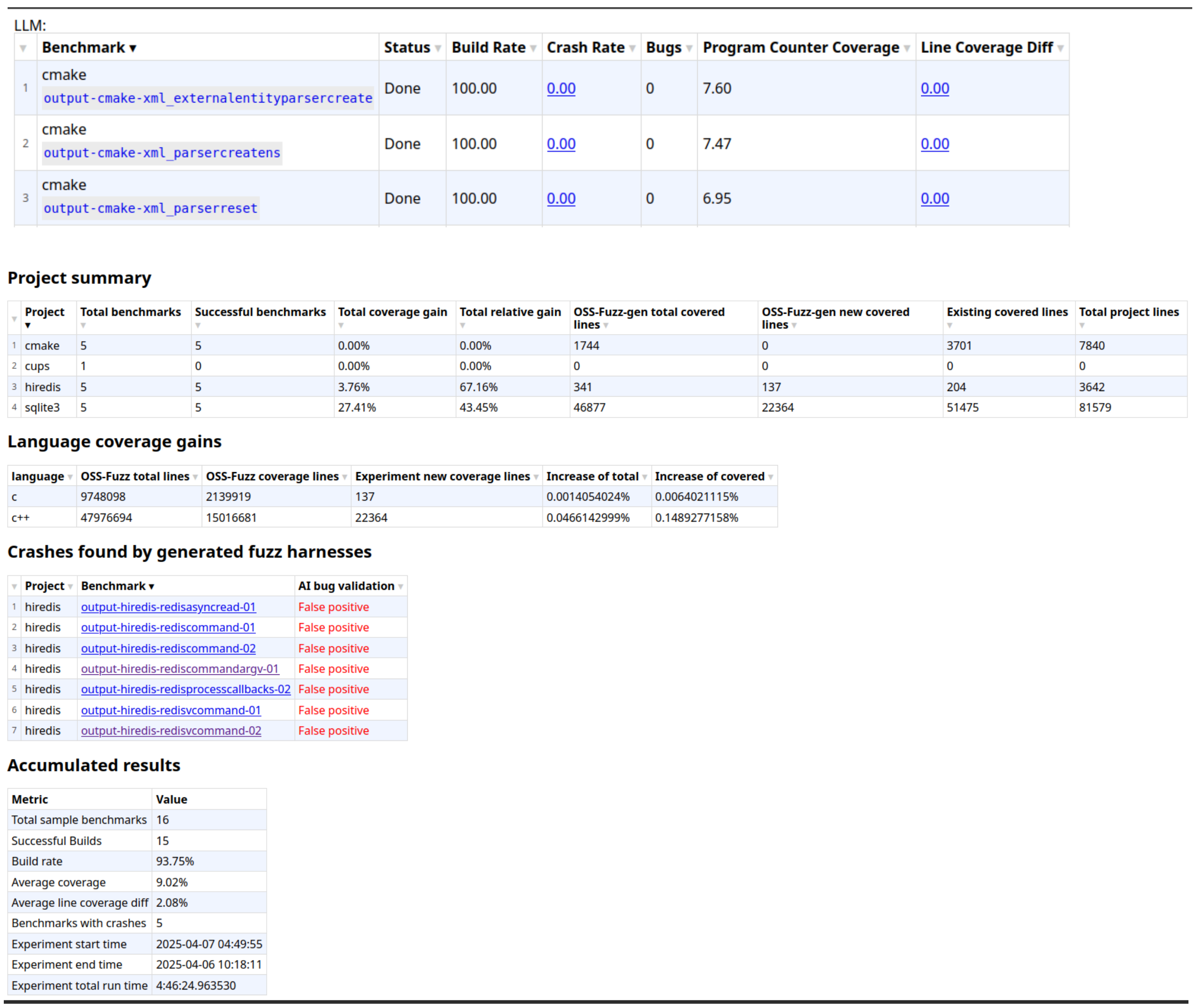
Proposed implementation: We will need to modify the `generate()` method of the `GenerateReport` in `report/web.py` to add an additional JSON write step:

```
self._write_index_html(benchmarks, accumulated_results, time_results,
                          projects, samples_with_bugs, coverage_language_gains)
self._write_index_json(benchmarks)
# ADD: An additional write step to write the unified JSON
self._write_unified_json(benchmarks, projects)
```

```
def _write_unified_json(self, benchmarks: List[Benchmark], projects: List[Project]):
   """Generate a unified JSON file with all benchmark and sample data."""
   unified_data = {
     project.name: {
       "project": project.name,
       "benchmarks": {},
       # And so on...
     } for project in projects
   }

   for benchmark in benchmarks:
       samples = self._results.get_samples(*self._results.get_results(benchmark.id))
       samples_data = []
       for sample in samples:
           sample_data = {
               "sample": sample.id,
               "status": sample.result.finished,
              # And so on...
           }
           samples_data.append(sample_data)
       # Calculate summary benchmark metrics

       unified_data[benchmark.project]["benchmarks"][benchmark.id] = {
           "samples": samples_data,
           # And so on...
       }
      # Calculate summary project metrics

   self._write('unified_data.json', json.dumps(unified_data, indent=2))
```

Main Index Page

Current index page UI layout:



LLM:

	Benchmark ▾	Status	Build Rate	Crash Rate	Bugs	Program Counter Coverage	Line Coverage Diff
1	cmake `output-cmake-xml_externalentityparsercreate`	Done	100.00	0.00	0	7.60	0.00
2	cmake `output-cmake-xml_parsercreatens`	Done	100.00	0.00	0	7.47	0.00
3	cmake `output-cmake-xml_parserreset`	Done	100.00	0.00	0	6.95	0.00

Project summary

	Project ▾	Total benchmarks	Successful benchmarks	Total coverage gain	Total relative gain	OSS-Fuzz-gen total covered lines	OSS-Fuzz-gen new covered lines	Existing covered lines	Total project lines
1	cmake	5	5	0.00%	0.00%	1744	0	3701	7840
2	cups	1	0	0.00%	0.00%	0	0	0	0
3	hiredis	5	5	3.76%	67.16%	341	137	204	3642
4	sqlite3	5	5	27.41%	43.45%	46877	22364	51475	81579

Language coverage gains

language	OSS-Fuzz total lines	OSS-Fuzz coverage lines	Experiment new coverage lines	Increase of total	Increase of covered
c	9748098	2139919	137	0.0014054024%	0.0064021115%
c++	47976694	15016681	22364	0.0466142999%	0.1489277158%

Crashes found by generated fuzz harnesses

	Project	Benchmark ▾	AI bug validation
1	hiredis	output-hiredis-redisasyncread-01	False positive
2	hiredis	output-hiredis-rediscommand-01	False positive
3	hiredis	output-hiredis-rediscommand-02	False positive
4	hiredis	output-hiredis-rediscommandargv-01	False positive
5	hiredis	output-hiredis-redisprocesscallbacks-02	False positive
6	hiredis	output-hiredis-redisvcommand-01	False positive
7	hiredis	output-hiredis-redisvcommand-02	False positive

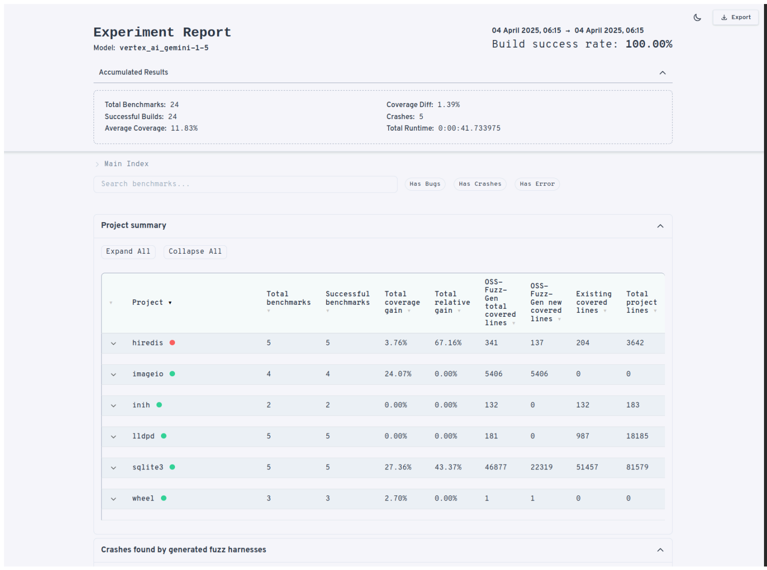
Accumulated results

Metric	Value
Total sample benchmarks	16
Successful Builds	15
Build rate	93.75%
Average coverage	9.02%
Average line coverage diff	2.08%
Benchmarks with crashes	5
Experiment start time	2025-04-07 04:49:55
Experiment end time	2025-04-06 10:18:11
Experiment total run time	4:46:24.963530

Implemented improvements for the prototype:

- [UI] Grouping of benchmarks to projects using a nested tables layout
- [UI] Dark mode for comfortable viewing at night
- [NAV] Navigable breadcrumbs to quickly move between pages
- [VIZ] Pie and bar graphs to visualise language coverage statistics
- [VIZ] Plots to compare the project line/function coverage between OSS-Fuzz-Gen and existing OSS-Fuzz coverage

Prototype index page UI layout:



Experiment Report
Model: vertex_ai_gemini-1-5

04 April 2025, 06:15 → 04 April 2025, 06:15
Build success rate: 100.00%

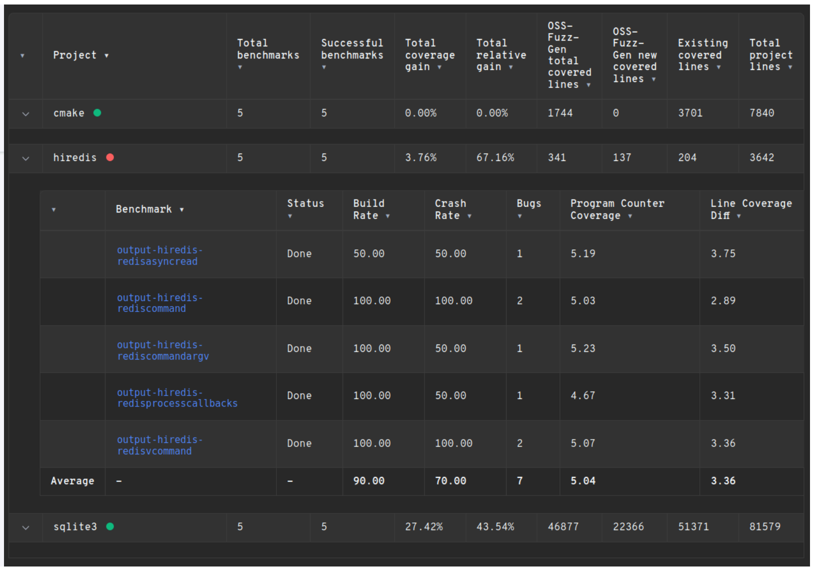
Accumulated Results

Total Benchmarks: 24	Coverage Diff: 1.39%
Successful Builds: 24	Crashes: 5
Average Coverage: 11.83%	Total Runtime: 0:00:41.733975

Project summary

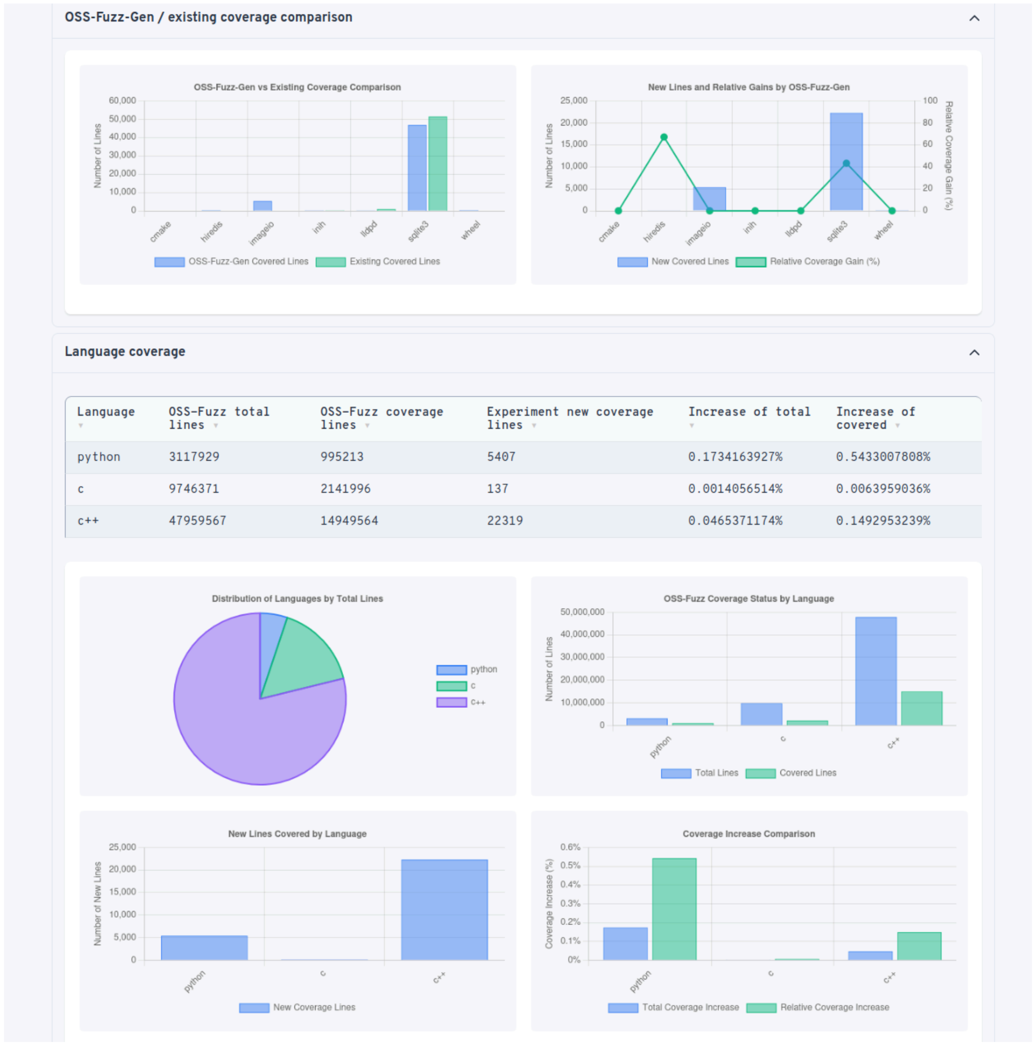
Project	Total benchmarks	Successful benchmarks	Total coverage gain	Total relative gain	OSS-Fuzz-Gen total covered lines	OSS-Fuzz-Gen new covered lines	Existing covered lines	Total project lines
hiredis 🔴	5	5	3.76%	67.16%	341	137	204	3642
imageio 🟢	4	4	24.07%	0.00%	5406	5406	0	0
inih 🟢	2	2	0.00%	0.00%	132	0	132	183
lldpd 🟢	5	5	0.00%	0.00%	181	0	987	18185
sqlite3 🟢	5	5	27.36%	43.37%	46877	22319	51457	81579
wheel 🟢	3	3	2.70%	0.00%	1	1	0	0

Crashes found by generated fuzz harnesses



Project	Total benchmarks	Successful benchmarks	Total coverage gain	Total relative gain	OSS-Fuzz-Gen total covered lines	OSS-Fuzz-Gen new covered lines	Existing covered lines	Total project lines
cmake 🟢	5	5	0.00%	0.00%	1744	0	3701	7840
hiredis 🔴	5	5	3.76%	67.16%	341	137	204	3642
sqlite3 🟢	5	5	27.42%	43.54%	46877	22366	51371	81579

Benchmark	Status	Build Rate	Crash Rate	Bugs	Program Counter Coverage	Line Coverage Diff
output-hiredis-redisasyncread	Done	50.00	50.00	1	5.19	3.75
output-hiredis-rediscommand	Done	100.00	100.00	2	5.03	2.89
output-hiredis-rediscommandargv	Done	100.00	50.00	1	5.23	3.50
output-hiredis-redisprocesscallbacks	Done	100.00	50.00	1	4.67	3.31
output-hiredis-redisvcommand	Done	100.00	100.00	2	5.07	3.36
Average -	-	90.00	70.00	7	5.04	3.36



Language	OSS-Fuzz total lines	OSS-Fuzz coverage lines	Experiment new coverage lines	Increase of total	Increase of covered
python	3117929	995213	5407	0.1734163927%	0.5433007808%
c	9746371	2141996	137	0.0014056514%	0.0063959036%
c++	47959567	14949564	22319	0.0465371174%	0.1492953239%

[FEAT] Search/filter functionality

A **working search functionality** that allows searching for information in all pages of the 3 templates: main index, benchmark, and trial/sample. Some possible filters:

- `ERR`: Whether a specific error happened
- `LANG`: Whether a specific programming language was used
- `CRASH`: Whether a trial crashed
- `COV`: Whether the coverage is less than/greater than/equal to `X`, where `X` can be a numerical value or the corresponding OSS-Fuzz coverage

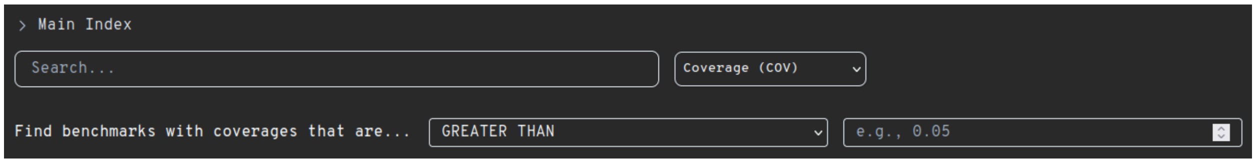
Filters should also be **combinable to form boolean expressions**. For example, it should be possible to search for trials that crashed `AND` had a coverage of less than `X` `OR` is a C++ project. This is potentially complex and can be a stretch task.

Proposed implementation:

With the unified JSON, the search task boils down to traversing the nested JSON structure and returning relevant, criteria-matching sections. The most robust solution for this would be to use JMESPath where simple string inputs are string-matching queries on all three layers (projects, benchmarks, samples) and filters are more complex template queries.

An example JMESPath query to search for benchmarks containing at least one sample with total coverage greater than 0.05 (equivalent to applying the `COV` filter with the operator `GREATER THAN` and the value `0.05`):

```
values(@)[?length(benchmarks.*.samples[] | [?total_coverage >
`0.05`]) > `0`].benchmarks
```


> Main Index
Search...
Coverage (COV)
Find benchmarks with coverages that are...
GREATER THAN
e.g., 0.05

```
function filterByCoverage() {
  const operator = document.getElementById('operator').value;
  const value = parseFloat(document.getElementById('covValue').value);
  return `values(@)[?length(benchmarks.*.samples[] | [?total_coverage
${operator} ${value}]) > 0].benchmarks`;
}
jmespath.search(unified_json, filterByCoverage());
```

[FEAT] Export to other file formats

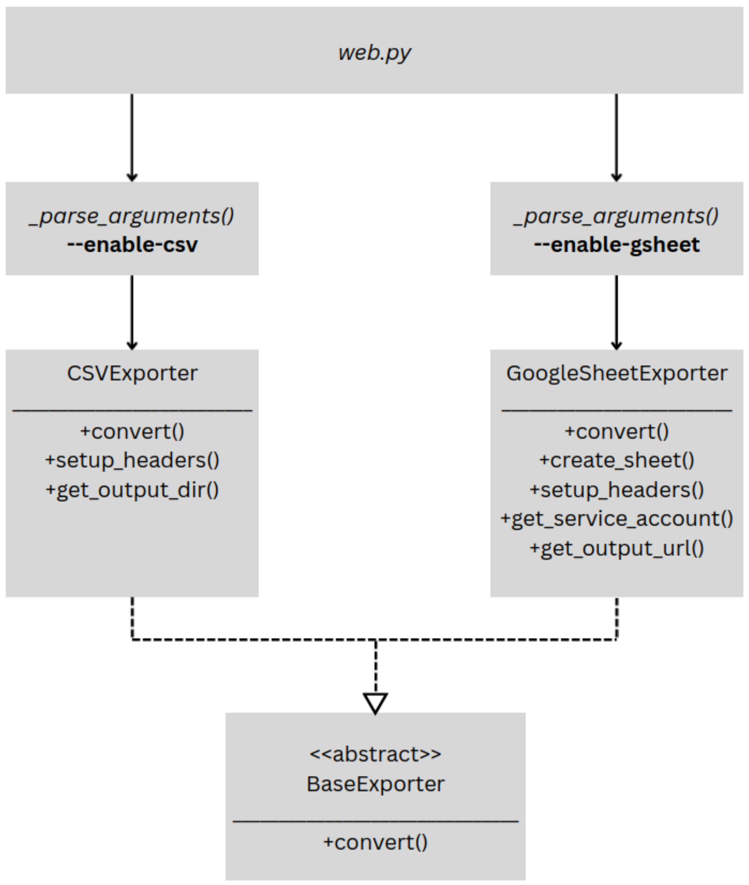
In particular, exporting to Google Sheet and CSV take the highest priority as they are the most repeatedly used formats for OSS-Fuzz-Gen.

Proposed implementation (CSV):

Add an optional `--enable-csv` flag to the argument parser in `web.py` to allow the user to opt into CSV export. Use a `CSVExporter` class during the report generation to generate an `experiment.csv` containing broadly the full experiment data. The user can then access it at `/<output-dir>/experiment.csv` by clicking on a download hyperlink in the report.

Proposed implementation (Google Sheet):

Add an optional `--enable-gsheet` flag to the argument parser in `web.py` to allow the user to opt into Google Sheet export. Similar to CSV, there can be a `GoogleSheetExporter` class that programmatically creates a new Google Sheet with the full experiment data (accessing the existing service account configuration for authentication) and returns the sheet URL, `https://docs.google.com/spreadsheets/d/{sheet.id}`. The URL can then be displayed on the report with a hyperlink, which opens the generated Google Sheet in a new browser tab.



*This is optional

(Optional) [FEAT] Subsection exports

This task refers to the ability to export only what the user currently sees on the page instead of the entire experiment data. For example, the user might want to export only the benchmark and sample data for the `redisCommand` function in `hiredis` instead of the full experiment data (which might be very, very large and contain other projects the user is not as interested in). The unified JSON should make this task easier to implement.

Due to its potential complexity and low priority, however, this is going to be an optional **stretch task.**

[NAV] Table of content

Display a dynamic **table of content** as a fixed-position side navigation drawer. The table of content should adapt accordingly to the page that's currently being viewed i.e. the index page should display sections, projects, and benchmarks while the benchmark page should display samples. As a stretch task, we can also use aggregated and average statistics to determine whether a specific sample is an outlier (so, interesting) on the table of content.

Proposed implementation:

We can use a generic hierarchy of BEM-inspired HTML classes to tag a specific element, then have a single shared JavaScript function to handle the dynamic rendering.

Index page:

```
Table of Content
├──Project Summary (.toc-section)
│   ├── hiredis (.toc-subsection)
│   │   ├──redisasyncread (.toc-item)
│   │   └──...
│   └── ...
├── OSS-Fuzz-Gen / existing coverage (.toc-section)
└── Language coverage (.toc-section)
```

Benchmark page:

```
Table of Content
├──redisasyncread (.toc-section)
│   ├──01 (.toc-subsection)
│   └──02 [Outlier] (.toc-subsection)
│   └── ...
```

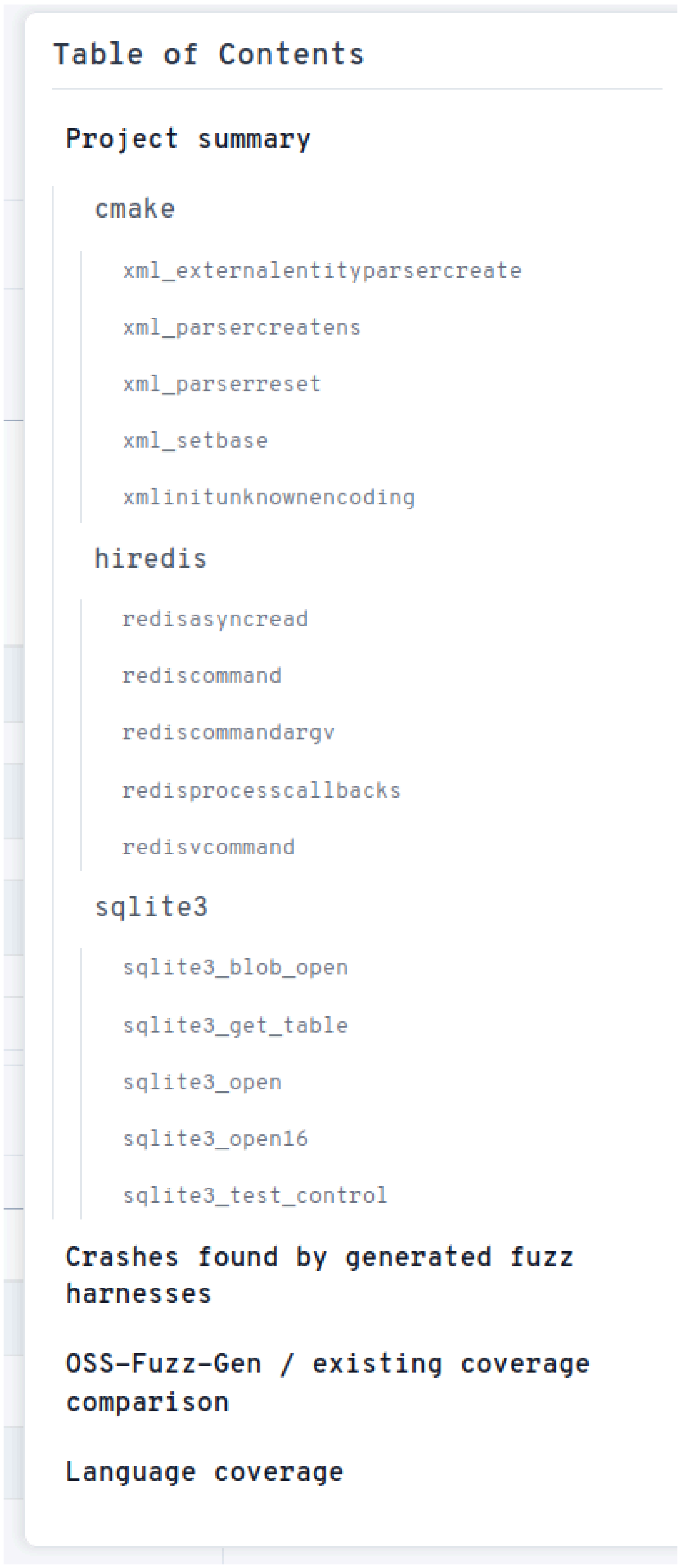


Table of Contents

Project summary

cmake

xml_externalentityparsercreate

xml_parsercreatens

xml_parserreset

xml_setbase

xmlinitunknownencoding

hiredis

redisasyncread

rediscommand

rediscommandargv

redisprocesscallbacks

redisvcommand

sqlite3

sqlite3_blob_open

sqlite3_get_table

sqlite3_open

sqlite3_open16

sqlite3_test_control

Crashes found by generated fuzz harnesses

OSS-Fuzz-Gen / existing coverage comparison

Language coverage

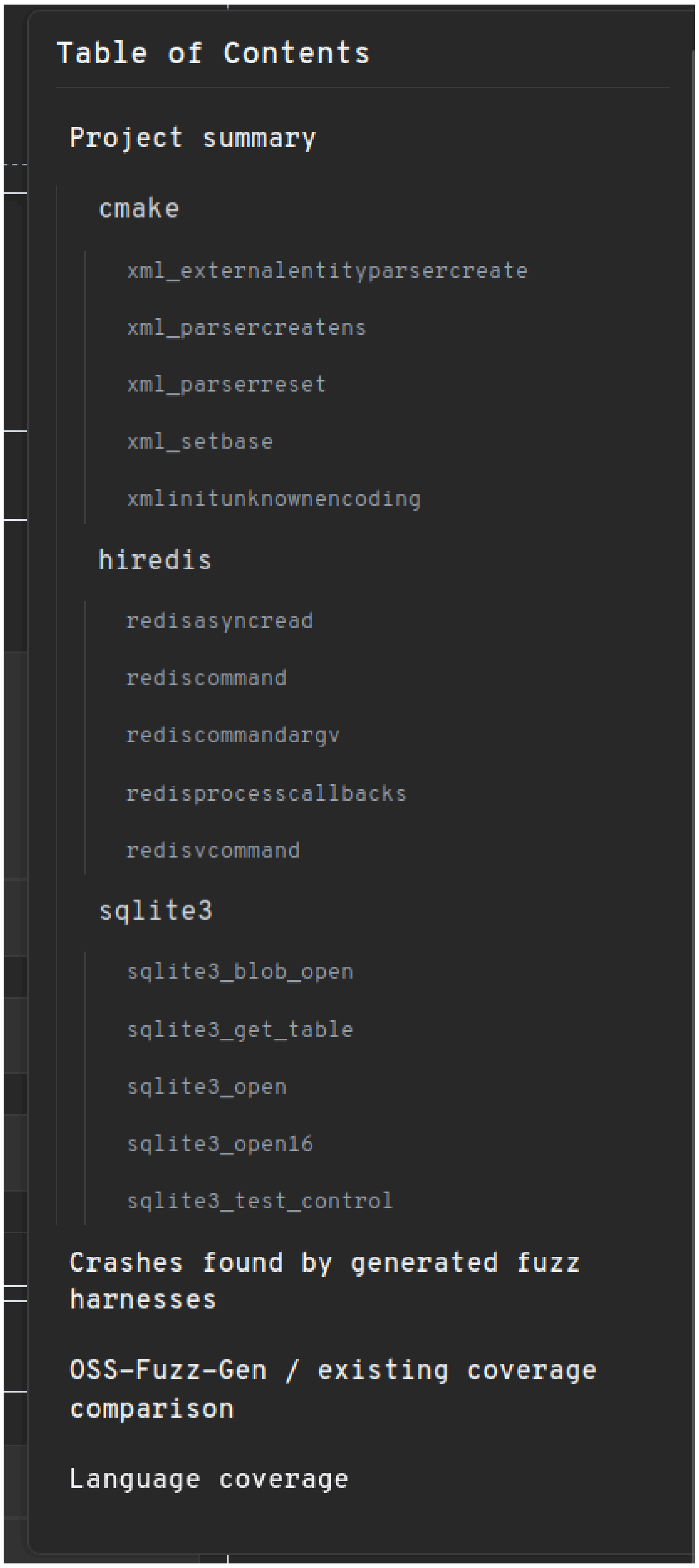


Table of Contents

Project summary

cmake

xml_externalentityparsercreate

xml_parsercreatens

xml_parserreset

xml_setbase

xmlinitunknownencoding

hiredis

redisasyncread

rediscommand

rediscommandargv

redisprocesscallbacks

redisvcommand

sqlite3

sqlite3_blob_open

sqlite3_get_table

sqlite3_open

sqlite3_open16

sqlite3_test_control

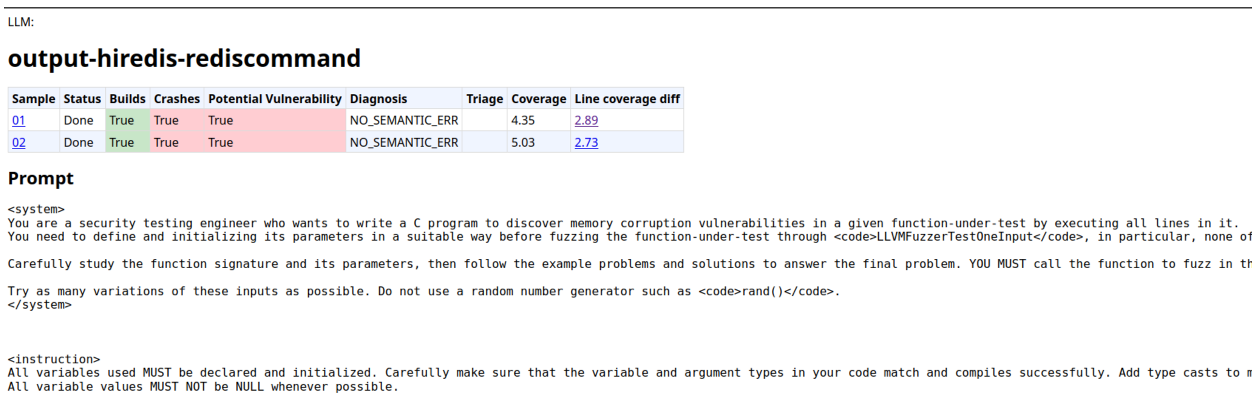
Crashes found by generated fuzz harnesses

OSS-Fuzz-Gen / existing coverage comparison

Language coverage

Benchmark Page

Current benchmark page UI layout:



LLM:

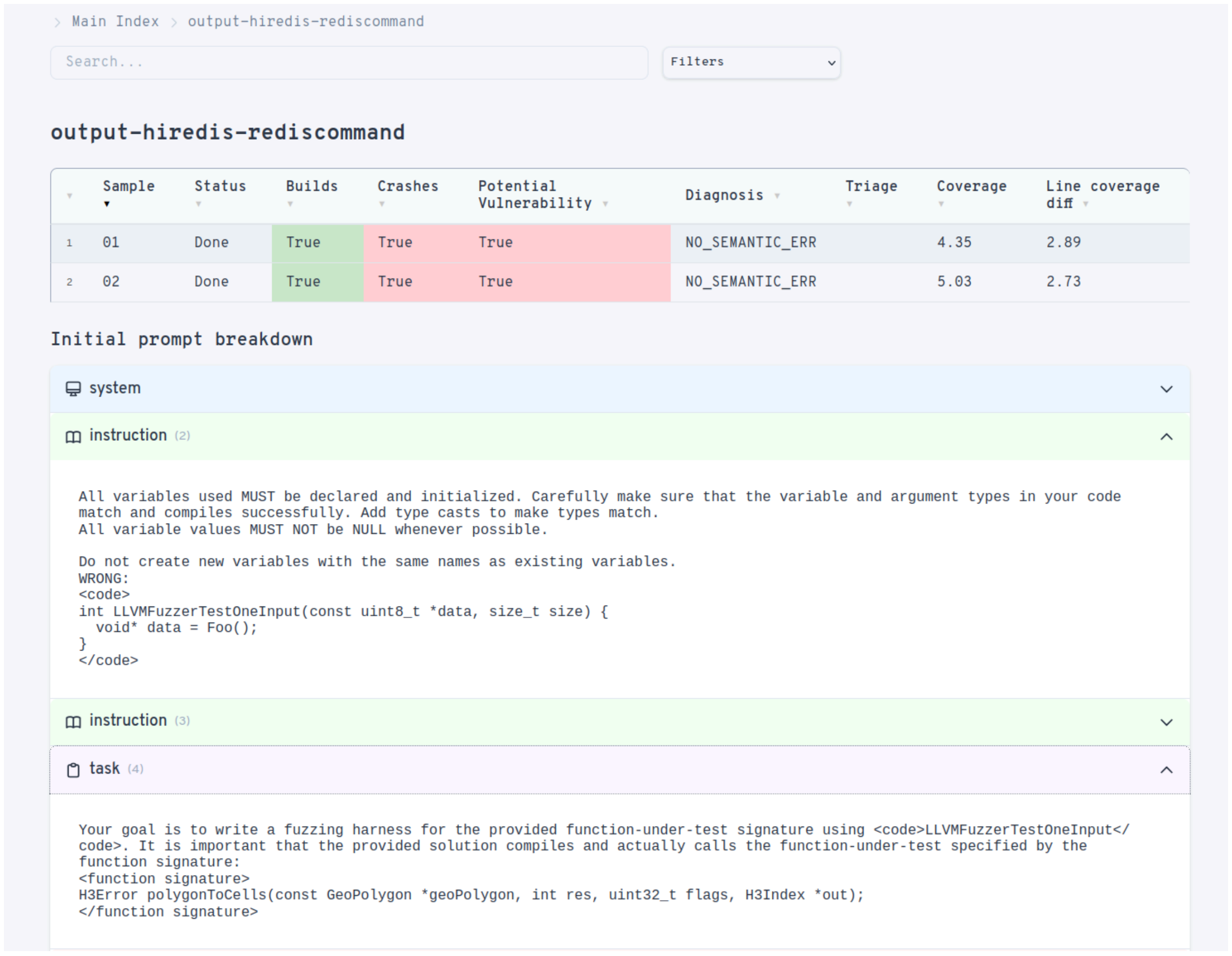
output-hiredis-rediscommand

Sample	Status	Builds	Crashes	Potential Vulnerability	Diagnosis	Triage	Coverage	Line coverage diff
01	Done	True	True	True	NO_SEMANTIC_ERR		4.35	2.89
02	Done	True	True	True	NO_SEMANTIC_ERR		5.03	2.73

Prompt

```
<system>
You are a security testing engineer who wants to write a C program to discover memory corruption vulnerabilities in a given function-under-test by executing all lines in it.
You need to define and initializing its parameters in a suitable way before fuzzing the function-under-test through <code>LLVMFuzzerTestOneInput</code>, in particular, none of

Carefully study the function signature and its parameters, then follow the example problems and solutions to answer the final problem. YOU MUST call the function to fuzz in th

Try as many variations of these inputs as possible. Do not use a random number generator such as <code>rand()</code>.
</system>


<instruction>
All variables used MUST be declared and initialized. Carefully make sure that the variable and argument types in your code match and compiles successfully. Add type casts to m
All variable values MUST NOT be NULL whenever possible.
```

Prototype benchmark page UI layout:



> Main Index > output-hiredis-rediscommand

Search...

Filters

output-hiredis-rediscommand

	Sample	Status	Builds	Crashes	Potential Vulnerability	Diagnosis	Triage	Coverage	Line coverage diff
1	01	Done	True	True	True	NO_SEMANTIC_ERR		4.35	2.89
2	02	Done	True	True	True	NO_SEMANTIC_ERR		5.03	2.73

Initial prompt breakdown

system

instruction (2)

```
All variables used MUST be declared and initialized. Carefully make sure that the variable and argument types in your code
match and compiles successfully. Add type casts to make types match.
All variable values MUST NOT be NULL whenever possible.

Do not create new variables with the same names as existing variables.
WRONG:
<code>
int LLVMFuzzerTestOneInput(const uint8_t *data, size_t size) {
  void* data = Foo();
}
</code>
```

instruction (3)

task (4)

```
Your goal is to write a fuzzing harness for the provided function-under-test signature using <code>LLVMFuzzerTestOneInput</
code>. It is important that the provided solution compiles and actually calls the function-under-test specified by the
function signature:
<function signature>
H3Error polygonToCells(const GeoPolygon *geoPolygon, int res, uint32_t flags, H3Index *out);
</function signature>
```

Implemented improvements for the prototype:

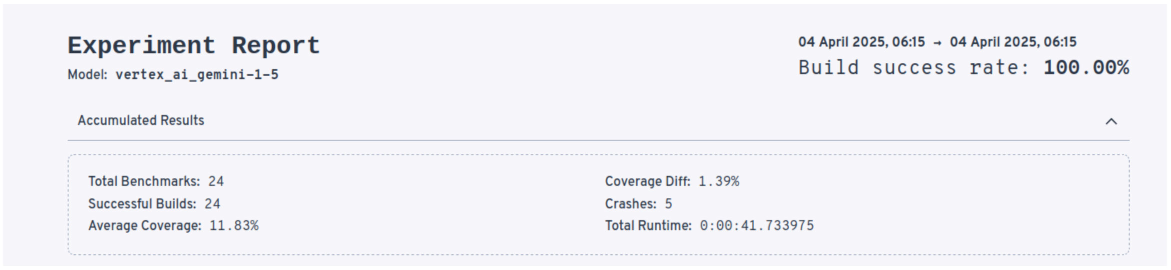
- [UI] Refined the table design for the samples table
- [UI] Extracted specific sections (`<system>`, `<instruction>`, `<task>`, `<solution>`) from the initial LLM prompt and presented them as colour-coded, order-preserving collapsible accordions

[UI] Refine structured initial prompt viewer

Additional tasks on the structured prompt viewer involve **adding more control buttons** to expand/collapse all accordions and adding language-specific **syntax highlighting** within `<code>` and `<function signature>` tags.

[AGG] Accumulated results header for benchmarks

We can have a header layout in `base.html` that automatically adapts to display relevant accumulated results for the overall experiment or the specific benchmark being viewed.



Experiment Report
Model: vertex_ai_gemini-1-5

04 April 2025, 06:15 → 04 April 2025, 06:15
Build success rate: **100.00%**

Accumulated Results

Total Benchmarks: 24	Coverage Diff: 1.39%
Successful Builds: 24	Crashes: 5
Average Coverage: 11.83%	Total Runtime: 0:00:41.733975

```
<div class="space-y-2">
    <p><span class="font-medium">Total Benchmarks:</span>
{{accumulated_results.total_runs}}</p>
    <p><span class="font-medium">Successful Builds:</span>
{{accumulated_results.compiles}}</p>
    <p><span class="font-medium">Average Coverage:</span>
{{accumulated_results.average_coverage |percent }}%</p>
</div>
<div class="space-y-2">
    <p><span class="font-medium">Coverage Diff:</span>
{{accumulated_results.average_line_coverage_diff |percent }}%</p>
    <p><span class="font-medium">Crashes:</span>
{{accumulated_results.crashes}}</p>
    <p><span class="font-medium">Total Runtime:</span>
{{time_results.total_run_time}}</p>
</div>
```

Currently, due to the ways the `Benchmark` and `AccumulatedResult` data classes are defined, `accumulated_results.average_coverage` here will not accurately refer to the average of all sample coverages but simply the highest coverage.

From the `generate()` function in `web.py`:

```
accumulated_results = self._results.get_macro_insights(benchmarks)
```

However, if we are on a specific benchmark's page, `benchmarks` only has one element (the benchmark itself), which leads to the aforementioned behaviour.

```
def get_macro_insights(self,
                        benchmarks: list[Benchmark]) -> AccumulatedResult:
   """Returns macro insights from the aggregated benchmark results."""
   accumulated_results = AccumulatedResult()
   for benchmark in benchmarks:
     accumulated_results.compiles += int(
         benchmark.result.build_success_rate > 0.0)
     accumulated_results.crashes += int(benchmark.result.found_bug > 0)
     accumulated_results.total_coverage += benchmark.result.max_coverage
     accumulated_results.total_runs += 1
     accumulated_results.total_line_coverage_diff += (
         benchmark.result.max_line_coverage_diff)
   return accumulated_results
```

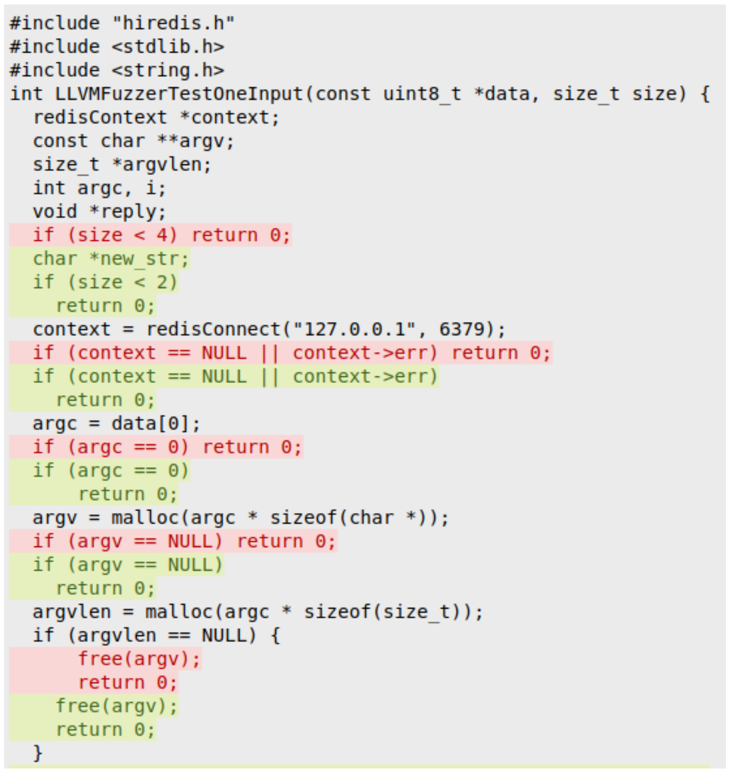
This is similar for other `accumulated_results` metrics as well.

Proposed implementation: Modify `get_macro_insights` itself to include special handling for the case where there is only one benchmark.

[FEAT] Fuzz target comparison tool

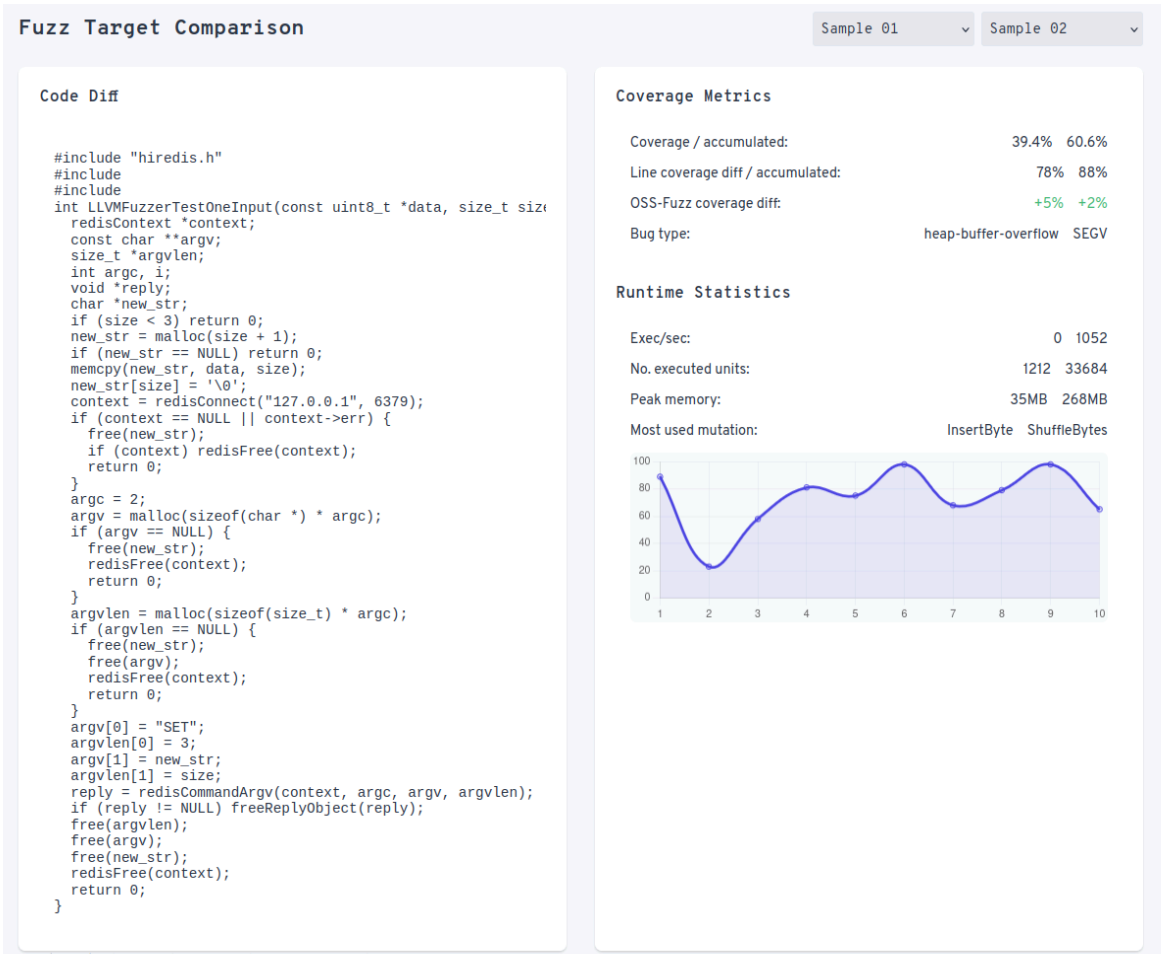
A feature to compare two or more different fuzz targets of the same function side-by-side. The comparison will include **highlighted final code differences** between the fuzz targets, their coverage / benchmark accumulated coverage ratios, as well as statistics extracted from their respective run logs (more details below). It will also compare the coverage of the selected fuzz target(s) **against their existing OSS-Fuzz coverages,** if applicable**.**

For code differences highlighting specifically, we can use the lightweight jsdiff script (via CDN injection). Here's an example of it applied to compare two different LLM-generated fuzz targets:



```
#include "hiredis.h"
#include <stdlib.h>
#include <string.h>
int LLVMFuzzerTestOneInput(const uint8_t *data, size_t size) {
  redisContext *context;
  const char **argv;
  size_t *argvlen;
  int argc, i;
  void *reply;
  if (size < 4) return 0;
  char *new_str;
  if (size < 2)
    return 0;
  context = redisConnect("127.0.0.1", 6379);
  if (context == NULL || context->err) return 0;
  if (context == NULL || context->err)
    return 0;
  argc = data[0];
  if (argc == 0) return 0;
  if (argc == 0)
      return 0;
  argv = malloc(argc * sizeof(char *));
  if (argv == NULL) return 0;
  if (argv == NULL)
    return 0;
  argvlen = malloc(argc * sizeof(size_t));
  if (argvlen == NULL) {
      free(argv);
      return 0;
    free(argv);
    return 0;
  }
```

Proposed layout design:



Fuzz Target Comparison

Code Diff

```
#include "hiredis.h"
#include
#include
int LLVMFuzzerTestOneInput(const uint8_t *data, size_t size
  redisContext *context;
  const char **argv;
  size_t *argvlen;
  int argc, i;
  void *reply;
  char *new_str;
  if (size < 3) return 0;
  new_str = malloc(size + 1);
  if (new_str == NULL) return 0;
  memcpy(new_str, data, size);
  new_str[size] = '\0';
  context = redisConnect("127.0.0.1", 6379);
  if (context == NULL || context->err) {
    free(new_str);
    if (context) redisFree(context);
    return 0;
  }
  argc = 2;
  argv = malloc(sizeof(char *) * argc);
  if (argv == NULL) {
    free(new_str);
    redisFree(context);
    return 0;
  }
  argvlen = malloc(sizeof(size_t) * argc);
  if (argvlen == NULL) {
    free(new_str);
    free(argv);
    redisFree(context);
    return 0;
  }
  argv[0] = "SET";
  argvlen[0] = 3;
  argv[1] = new_str;
  argvlen[1] = size;
  reply = redisCommandArgv(context, argc, argv, argvlen);
  if (reply != NULL) freeReplyObject(reply);
  free(argvlen);
  free(argv);
  free(new_str);
  redisFree(context);
  return 0;
}
```

Coverage Metrics

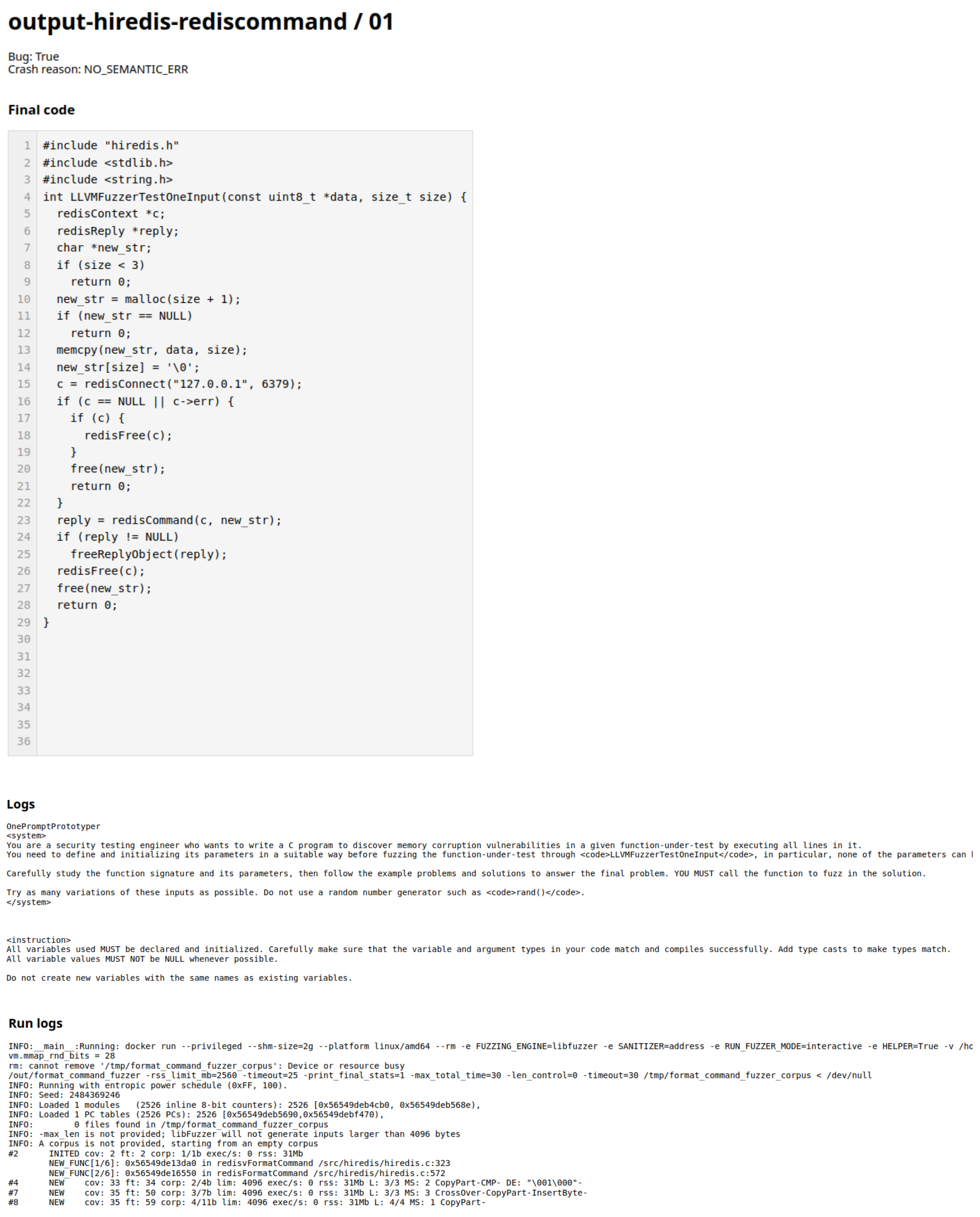
Coverage / accumulated:	39.4%	60.6%
Line coverage diff / accumulated:	78%	88%
OSS-Fuzz coverage diff:	+5%	+2%
Bug type:	heap-buffer-overflow	SEGV

Runtime Statistics

Exec/sec:	0	1052
No. executed units:	1212	33684
Peak memory:	35MB	268MB
Most used mutation:	InsertByte	ShuffleBytes

Trial/Sample Page

Current benchmark page UI layout:



output-hiredis-rediscommand / 01

Bug: True
Crash reason: NO_SEMANTIC_ERR

Final code

```c
#include "hiredis.h"
#include <stdlib.h>
#include <string.h>
int LLVMFuzzerTestOneInput(const uint8_t *data, size_t size) {
  redisContext *c;
  redisReply *reply;
  char *new_str;
  if (size < 3)
    return 0;
  new_str = malloc(size + 1);
  if (new_str == NULL)
    return 0;
  memcpy(new_str, data, size);
  new_str[size] = '\0';
  c = redisConnect("127.0.0.1", 6379);
  if (c == NULL || c->err) {
    if (c) {
      redisFree(c);
    }
    free(new_str);
    return 0;
  }
  reply = redisCommand(c, new_str);
  if (reply != NULL)
    freeReplyObject(reply);
  redisFree(c);
  free(new_str);
  return 0;
}
```

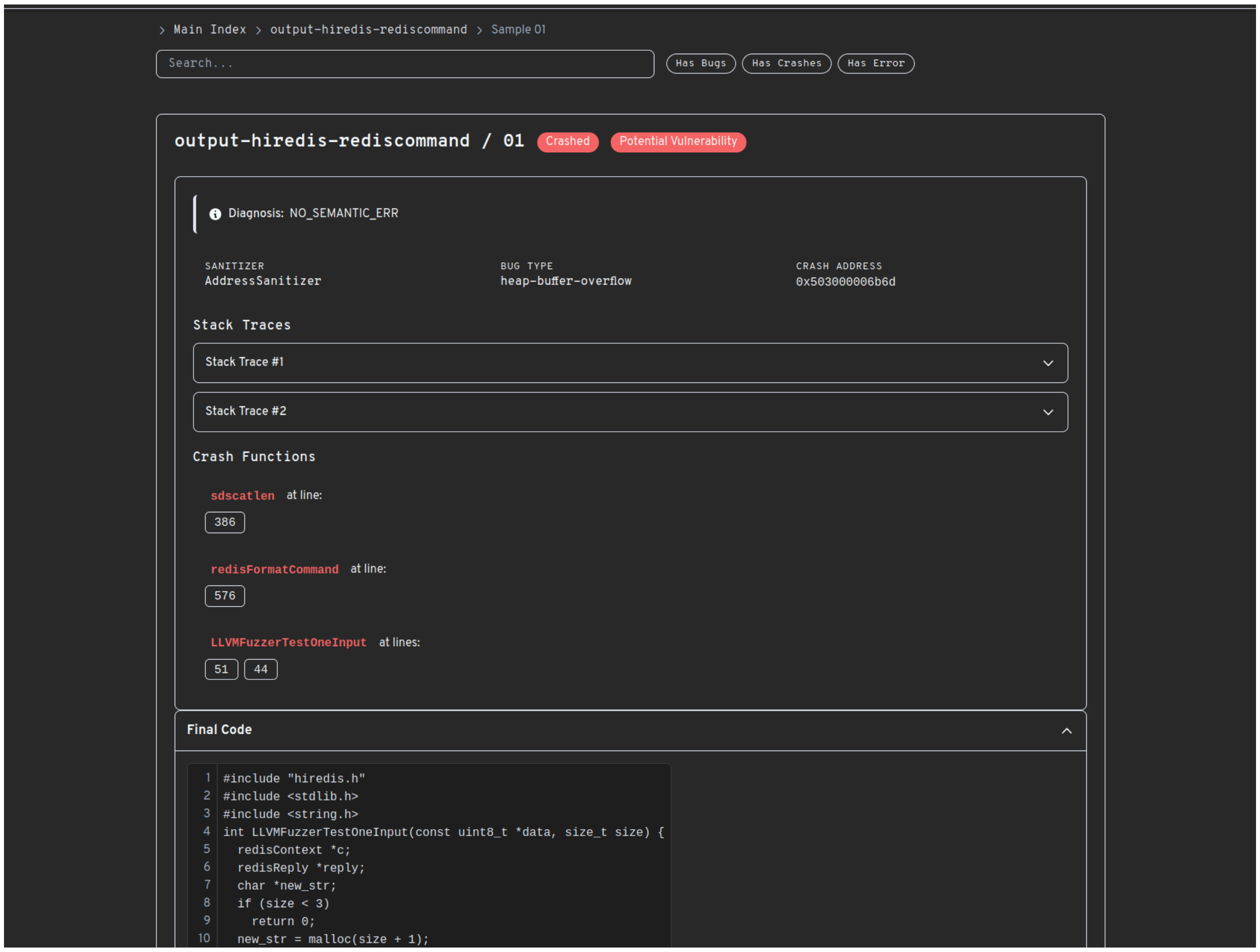
Logs

```
OnePromptPrototyper
<system>
You are a security testing engineer who wants to write a C program to discover memory corruption vulnerabilities in a given function-under-test by executing all lines in it.
You need to define and initializing its parameters in a suitable way before fuzzing the function-under-test through <code>LLVMFuzzerTestOneInput</code>, in particular, none of the parameters can

Carefully study the function signature and its parameters, then follow the example problems and solutions to answer the final problem. YOU MUST call the function to fuzz in the solution.

Try as many variations of these inputs as possible. Do not use a random number generator such as <code>rand()</code>.
</system>



<instruction>
All variables used MUST be declared and initialized. Carefully make sure that the variable and argument types in your code match and compiles successfully. Add type casts to make types match.
All variable values MUST NOT be NULL whenever possible.

Do not create new variables with the same names as existing variables.
```

Run logs

```
INFO:__main__:Running: docker run --privileged --shm-size=2g --platform linux/amd64 --rm -e FUZZING_ENGINE=libfuzzer -e SANITIZER=address -e RUN_FUZZER_MODE=interactive -e HELPER=True -v /hc
vm.mmap_rnd_bits = 28
rm: cannot remove '/tmp/format_command_fuzzer_corpus': Device or resource busy
/out/format_command_fuzzer -rss_limit_mb=2560 -timeout=25 -print_final_stats=1 -max_total_time=30 -len_control=0 -timeout=30 /tmp/format_command_fuzzer_corpus < /dev/null
INFO: Running with entropic power schedule (0xFF, 100).
INFO: Seed: 2484369246
INFO: Loaded 1 modules   (2526 inline 8-bit counters): 2526 [0x56549deb4cb0, 0x56549deb568e),
INFO: Loaded 1 PC tables (2526 PCs): 2526 [0x56549deb5690,0x56549debf470),
INFO:        0 files found in /tmp/format_command_fuzzer_corpus
INFO: -max_len is not provided; libFuzzer will not generate inputs larger than 4096 bytes
INFO: A corpus is not provided, starting from an empty corpus
#2      INITED cov: 2 ft: 2 corp: 1/1b exec/s: 0 rss: 31Mb
        NEW_FUNC[1/6]: 0x56549de13da0 in redisvFormatCommand /src/hiredis/hiredis.c:323
        NEW_FUNC[2/6]: 0x56549de16550 in redisFormatCommand /src/hiredis/hiredis.c:572
#4      NEW    cov: 33 ft: 34 corp: 2/4b lim: 4096 exec/s: 0 rss: 31Mb L: 3/3 MS: 2 CopyPart-CMP- DE: "\001\000"-
#7      NEW    cov: 35 ft: 50 corp: 3/7b lim: 4096 exec/s: 0 rss: 31Mb L: 3/3 MS: 3 CrossOver-CopyPart-InsertByte-
#8      NEW    cov: 35 ft: 59 corp: 4/11b lim: 4096 exec/s: 0 rss: 31Mb L: 4/4 MS: 1 CopyPart-
```

Prototype benchmark page UI layout (dark mode):



output-hiredis-rediscommand / 01

Diagnosis: NO_SEMANTIC_ERR

SANITIZER	BUG TYPE	CRASH ADDRESS
AddressSanitizer	heap-buffer-overflow	0x503000006b6d

Stack Traces

Crash Functions

sdscatlen at line: 386

redisFormatCommand at line: 576

LLVMFuzzerTestOneInput at lines: 51 44

```
#include "hiredis.h"
#include <stdlib.h>
#include <string.h>
int LLVMFuzzerTestOneInput(const uint8_t *data, size_t size) {
  redisContext *c;
  redisReply *reply;
  char *new_str;
  if (size < 3)
    return 0;
  new_str = malloc(size + 1);
```

Implemented improvements for the prototype:

- [UI] Structured layout into collapsible accordion sections
- [UI] Badges to highlight bug status and whether there are any potential vulnerability
- [FEAT] Extracted Semantic Analyser insights
- [VIZ] The coverage report, previously discoupled from the trial/sample page, is now embedded into this page via an `iframe` for more focused viewing

[UI] Refine layout for the trial/sample page

We can add a **copy button** for the user to copy the final code output, as well as a button to **export a reproduction script** for reproducing the coverage results in OSS-Fuzz. The path to the sample's **target_binary** can also be displayed with the Semantic Analyser insights.

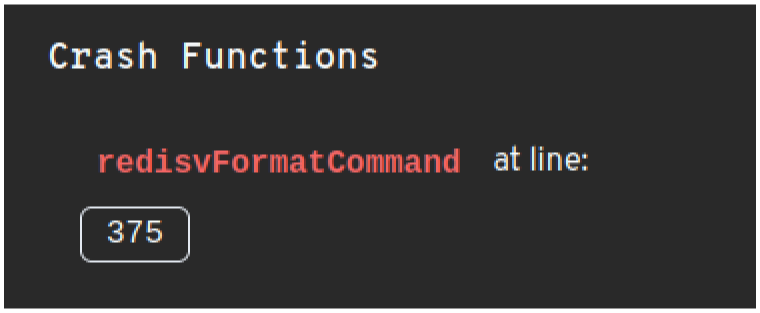
More plots can be added to compare the coverage performance between the LLM-generated fuzz target and any existing OSS-Fuzz fuzz target.

SImilar to the benchmark page, the logs can **display accordions** showing extracted LLM chat history sections. There should also be **control buttons** to expand/collapse all sections. Additionally, the run logs can be **formatted** with important logs highlighted or color-coded:

- `INFO` (yellow)
- `ERROR` (red)
- `STAT` (chartreuse green)
- `EVENT` logs: Lines starting with "#" containing `NEW` or `REDUCE` (light grey)

[FEAT] Link crash functions to coverage report

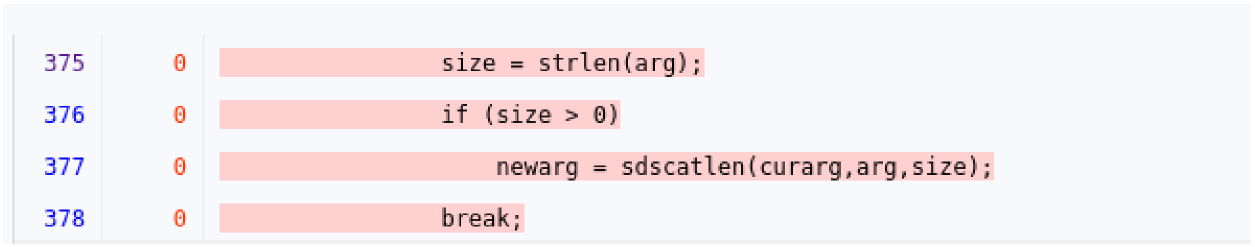
The aim of this task is to make it easier to navigate directly to the crashed lines in the coverage report. After the user clicks on a crash function's line number, for example `redisvFormatCommand` at line: 375, they should be shown the exact section in the coverage report.



Crash Functions

redisvFormatCommand at line:

375



```
375    0            size = strlen(arg);
376    0            if (size > 0)
377    0                newarg = sdscatlen(curarg,arg,size);
378    0            break;
```

One way to do this would be simply changing the URL of the coverage report `iframe` to the corresponding function/file path i.e. `sample/output-hiredis-rediscommand/coverage/02/linux/src/hiredis/hiredis.c.html#L375`, however only if we know the relevant file the function is located in (`hiredis.c`).

Fortunately, the stack traces contain this file path. Unfortunately, the current crash functions, which are determined through parsing stack traces in the `_parse_func_from_stacks` method in `semantic_analyzer.py`, don't. It only has the function name and the line numbers in an array.

```
return {
       func_name: list(line_numbers)
       for func_name, line_numbers in func_info.items()
   }
```

Proposed implementation: Modify the `_parse_func_from_stacks` method to record both the array of line numbers and the relevant file name/path for each crashed function.

[AGG] Run log extraction

The following should be extracted for every sample, displayed on their summary alongside the Semantic Analyser insights, and aggregated in the parent benchmark page:

- Execution: `stat::number_of_executed_units`,
- Performance: `stat::average_exec_per_sec`
- Memory: `stat::peak_rss_mb`
- Edge coverage: Final `cov:` value
- Number of features discovered: Final `ft:` value
- Types of mutations, for example `InsertRepeatedBytes`
- The fuzzing engine, e.g. `FUZZING_ENGINE=libfuzzer`
- Corpus statistics, e.g. `INFO: seed corpus: files: 11 min: 11b max: 749b total: 1809b rss: 31Mb`

Proposed implementation: Begin with simple string matching and RegEx, then iterate as needed.

[VIZ] Log events visualisation

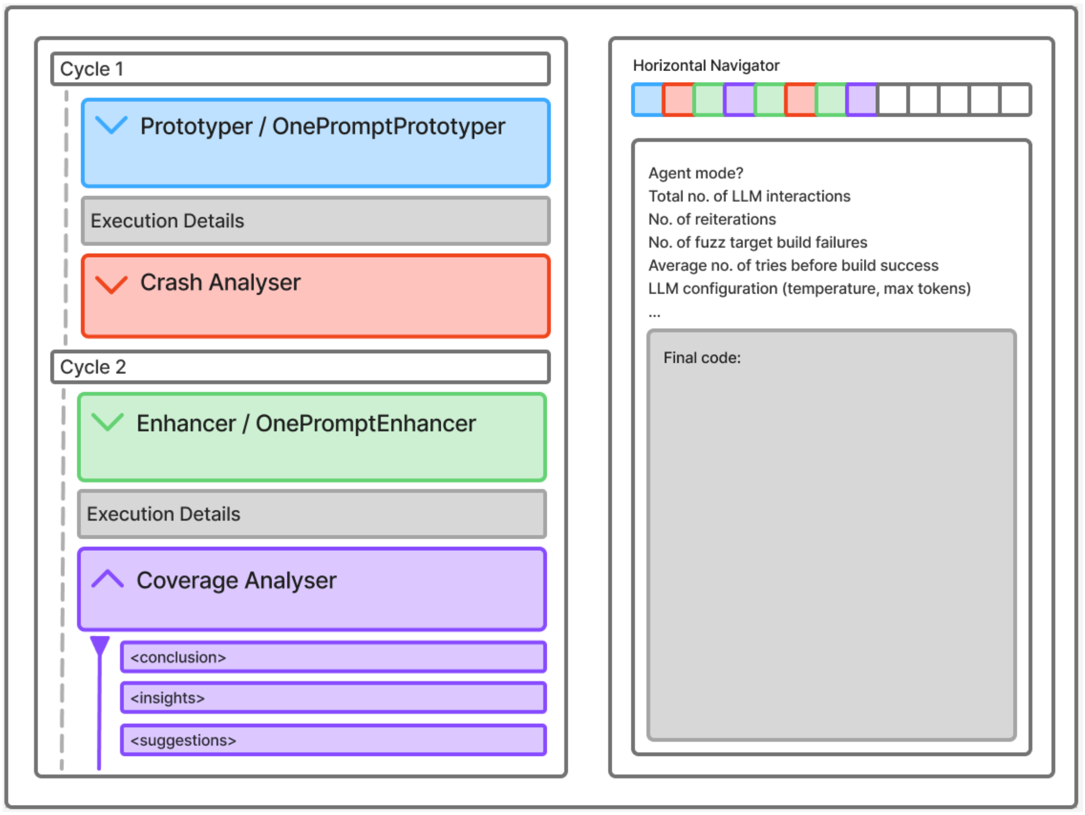
Event types visualiser from the run logs, of which there are two: `NEW` and `REDUCE`. Specifically, we can add a per-iteration trend graph showing how the coverage (`cov:`), feature count (`ft:`), number of mutation operations (`MS`), and specific mutations applied evolves at each iteration, as well as an event distribution graph showing the frequency of `NEW` and `REDUCE` events as well as mutation type frequencies.

(Optional) [FEAT] LLM interaction explorer

The distinguishing feature of OSS-Fuzz-Gen is its use of LLMs to generate fuzz targets. As such, these interactions are interesting to researchers. It would make sense to have a dedicated explorer page for each fuzz target to investigate how the various LLM interactions led to the final code output.

Due to its potential complexity, this will be an **optional stretch task** to be explored after the main tasks have been completed.

Proposed layout design:


Cycle 1
Prototyper / OnePromptPrototyper
Execution Details
Crash Analyser
Cycle 2
Enhancer / OnePromptEnhancer
Execution Details
Coverage Analyser
<conclusion>
<insights>
<suggestions>
Horizontal Navigator
Agent mode?
Total no. of LLM interactions
No. of reiterations
No. of fuzz target build failures
Average no. of tries before build success
LLM configuration (temperature, max tokens)
...
Final code:

Milestones

I propose that the project should be considered successful if all of the below **three** main milestones are completed by November 10, 18:00 UTC.

I. Milestone 1 (Core UI)

A. Index Page

- ☐ Table of content for large experiment sizes

- ☐ Unified experiment JSON
- ☐ Working search functionality (JMESPath integration)
- ☐ Working basic search filters
- ☐ Working export to CSV
- ☐ Working export to Google Sheet

B. Benchmark Page

- ☐ Accumulated results header correctly adapted for benchmarks
- ☐ Interactive tool for comparing fuzz targets of the same function
- ☐ Refined existing structured initial prompt viewer

C. Trial/Sample Page

- ☐ Final code copy and extract `oss-fuzz` reproduction script buttons
- ☐ Plots for fuzz target coverage comparisons
- ☐ Crash functions are linked to sections in the coverage report
- ☐ Logs are properly colour-coded and formatted for display

II. Milestone 2 (Log visualisation)

A. Log extraction

- ☐ Basic log extractions (`INFO`, `ERROR`, `STAT`, `EVENT`)
- ☐ Comprehensive log extractions (execution, performance, memory, edge coverage, number of discovered features, types of mutation, etc)
- ☐ Comprehensive log extractions are displayed on the respective trial/sample page

B. Log events visualisation

- ☐ Per-iteration plot showing how the coverage (`cov:`) and feature count (`ft:`) evolve at each iteration
- ☐ Per-iteration plot showing how the number of mutation operations (`MS`) and specific mutation applied evolves at each iteration
- ☐ Event distribution plot showing the frequency of `NEW` and `REDUCE` events for that particular trial/sample
- ☐ Event distribution plot showing the mutation type frequencies

III. Milestone 3 (Experiment workflow)

- ☐ The CI workflow checks the build status every polling interval
- ☐ The polling interval (default 10 minutes) is configurable using a GitHub comment by maintainers, similar to `/gcbrun`
- ☐ The CI workflow automatically prints relevant paths alongside relevant information as a GitHub comment and tags relevant maintainers

IV. Stretch Tasks

These are optional tasks that I will look into implementing **only after the main milestones have been completed.**

- ☐ Basic implementation of the LLM interaction explorer
- ☐ Combinable boolean expression filtering for the search functionality
- ☐ Subsection exports
- ☐ Table of content with outlier detection

Project Timeline

Present - May 07	**1 month**	I will be experimenting more with the JMESPath search functionality using this repo. There were also some general quirks encountered during my prototyping process that I'd like to raise issues for in the OSS-Fuzz-Gen repo and eventually resolve before the official coding period.
May 08 - June 01 (Community Bonding)	**1 month**	Revamp current prototype according to my mentor's feedback and refine existing prototyped features. I will also refactor the `template` folder by putting each of its three main templates into their own folder with related CSS and JavaScript files and modify the report generation pipeline accordingly.
June 02 - June 08 (Reduced availability)	**1 week**	I'll be preparing for my end-of-semester exams so I'll have reduced availability for this week. - **Index page:** Table of content - **Index page:** JSON consolidation code
June 09 - June 22	**2 weeks**	- **Index page**: Search functionality - **Index page**: Filter functionality
June 23 - June 29	**1 week**	- Working export of the report to CSV - Working export of the report to Google Sheet

June 30 - July 13 (Increased availability)	**2 weeks**	- **Benchmark page**: Accumulated results header - **Benchmark page:** Fuzz target comparison tool - **Benchmark page:** Refined prompt viewer - **Trial/sample page**: Refined layout - **Trial/sample page**: More plots for a more comprehensive fuzz target coverage comparison - **Trial/sample page:** Linking crash functions to coverage report Due to the increased availability I have for these two weeks, I will also work on these additional tasks: - **Logs:** Log extraction and aggregation code - **Logs**: Basic log extraction
		Milestone 1: Core UI reached
		Submit mid term evaluation
July 14 - July 25 (Increased availability)	**1.5 week**	- **Trial/sample page**: UI to display extracted log information for that particular fuzz target - **Logs:** Comprehensive log extraction - **Logs**: Log aggregation for the benchmark page Due to the increased availability I have for this week, I will also work on these additional tasks: - **Logs:** Log events extraction code - **Logs**: Per-iteration log events visualisation graphs - **Logs**: Log event distribution graphs
		Milestone 2: Log Visualisation reached
July 26 - August 18	**3 weeks**	Since I'm less familiar with CI workflows than with UI, I've allocated extra time to work on these two features. - **Experiment workflow:** Build status polling code - **Experiment workflow:** Automatic link printing code

August 19 - August 24	**6 days**	- **Experiment workflow:** Test all the CI workflow improvements, ensuring that there are no errors and unaccounted edge cases.
		Milestone 3: Experiment Workflow reached
August 25 - September 01	**1 week**	This is a buffer week for general code polishing, writing documentation, and addressing any gaps in implementation that've been left so far.
		Submit final evaluation
After September 01		If all milestones have been reached at this stage, I will use this time for the optional stretch tasks. Afterwards, I'd like to continue contributing to OSS-Fuzz-Gen by maintaining the report-related features that were implemented during Google Summer of Code and extending it with more features.

About Me

I'm a third-year student majoring in Mathematics and Computer Science at the University of Auckland, NZ. I was introduced to Google Summer of Code and specifically to open-source security by someone who worked on the OSV.dev project in Sydney. I have strong interests across all areas of web development. Last year, I was an open-source contributor with Tuono, made my own experimental frontend framework building on React/JSX, and dabbled in Ell Studio and Neuronpedia (which exposed me to the concept of LLM explorer UIs!). I've also previously used Python in a web development context while building a mapping visualisation tool (with Flask and JInja) for an Auckland-based non-profit client in community law last year.

Prior Contributions

In setting up the background for this proposal, I extended the existing local report compilation process to allow hot-reloading using a similar watcher system as outlined in the proposed experiment workflows improvements - this has helped significantly in quickly developing the prototype. I also implemented a minor colour fix to improve readability for a regular contributor with red-green colour-blindness.

Why OSS-Fuzz-Gen?

I didn't even know what code fuzzing was before OSS-Fuzz-Gen and that it's an active area of research, though the *idea* of it certainly occurred to me i.e. 'Why sit there and think up edge cases when you can just bombard the function with a bunch of random inputs and see how it goes?'. It's good to be able to put a name to the concept. This project specifically focuses on UI, which is my strongest point, but also involves something I'm less familiar with (experiment workflows) making it an interesting blend of familiar and novel topics. There's also a certain challenge even to the UI aspect: unlike most of my previous work, the OSS-Fuzz-Gen report shouldn't rely on heavy, complicated JavaScript dependencies like React because the reports are primarily uploaded to the cloud.

So, ideally, no `node_modules`.

This has forced me to think 'beyond my box' for tools that don't require complicated setups as to not add more work for my mentor and other OSS-Fuzz-Gen maintainers in the future, yet also powerful enough to create efficient streamlined UIs. I eventually came around to Alpine.js and actually really enjoyed using it to build the prototype.

Finally, the most pragmatic factor in my decision to choose OSS-Fuzz-Gen is the close geographic/time zone difference between Australia and NZ making it easier to communicate with my mentor. I greatly appreciate his insights and our back-and-forth during the drafting of this proposal as well as during GitHub discussions!

Availability

During most of the official coding phase, I will be able to spend roughly 20 hours per week working on the project. However, from July 01 to July 18, I will be available for 30-40 hours per week as I'll be having my inter-semester break then.