Analysis, Modeling and Prediction of Program Performance Based On Recent Testing Techniques

Analýza, modelování a predikce výkonu programu založená na testovacích metodách

---

Matúš Liščinský

Supervisor: **Doc. Mgr. Adam Rogalewicz, Ph.D.**

Motivation I

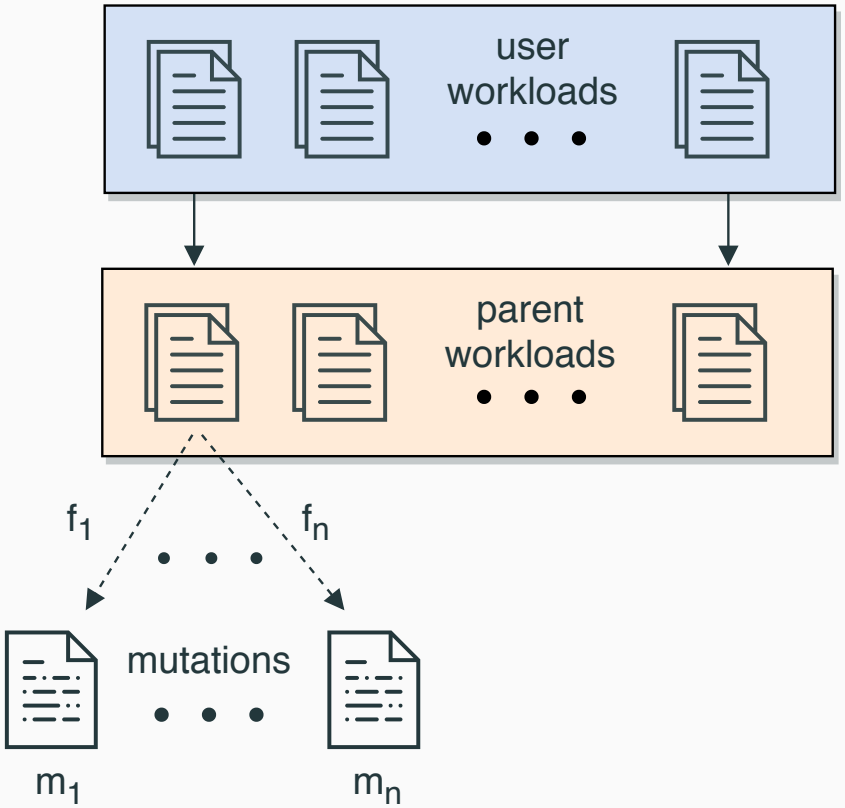
- **Perun** (Performance Versioning System and tool suite)
  - Management of **performance** during development.
- But, performance heavily depends on input workloads.
  - Perun takes user defined workload as input, and measures performance profile.

Motivation I

- **Perun** (Performance Versioning System and tool suite)
  - Management of **performance** during development.
- But, performance heavily depends on input workloads.
  - Perun takes user defined workload as input, and measures performance profile.
- Is it effective? Will the workloads trigger performance change?
  - Fuzzing was already successfully applied for functional testing.
  - What if we tune **fuzzing** techniques to find workload triggering performance issues ?

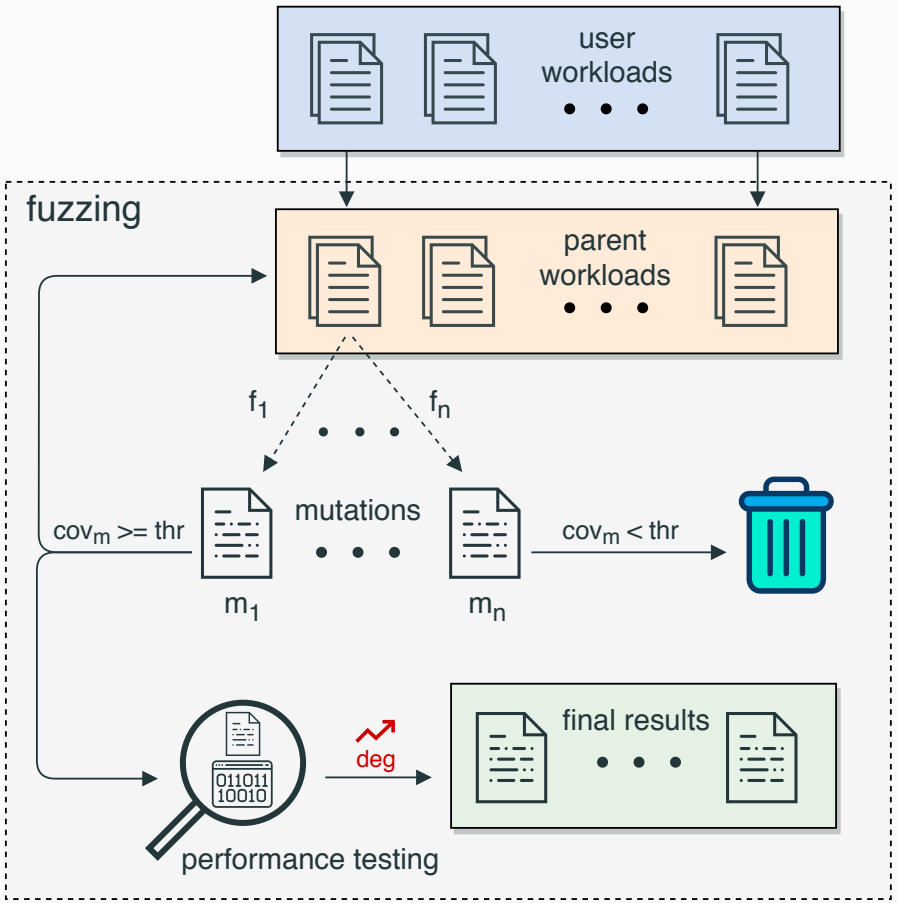
Performance Fuzzing

- **Goal:** Find new workloads triggering performance changes.
- User defined workloads serve as **initial samples** (seeds).
- Fuzzing gradually generates new workloads using **mutations**.



Performance Fuzzing Algorithm

- Parents are evaluated by **two criteria**:
  - Coverage of code: **fast**
  - Performance analysis result (by Perun): **precise**, but **slow**

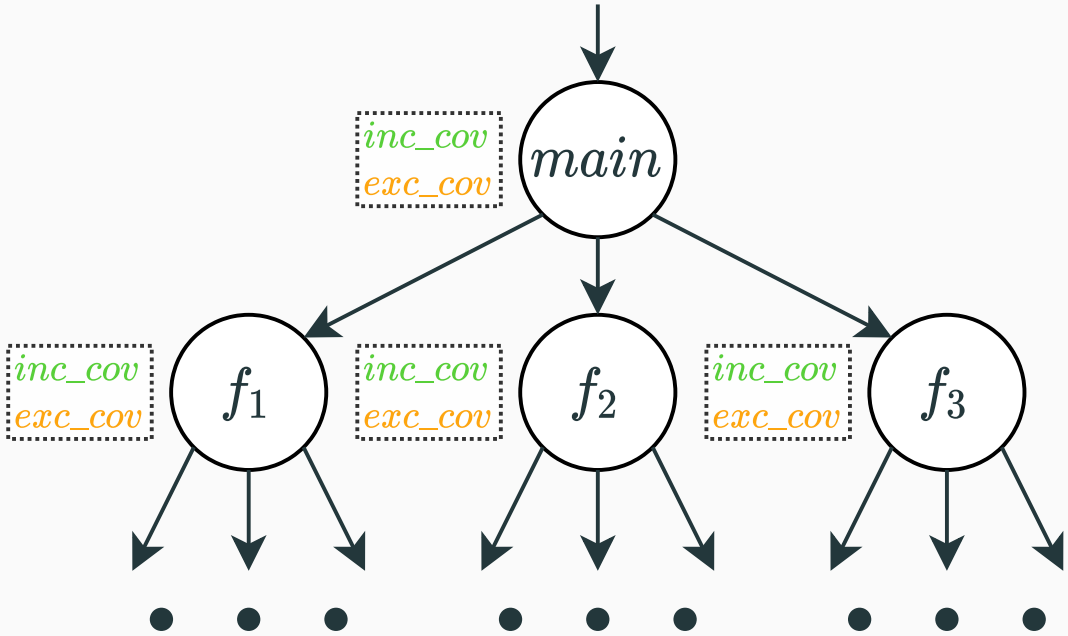


Motivation II

- Coverage indicator: executed LOC of target application
  - rough approximation (one single value)
  - local maximum can be easily lost, i.e. the change will not be detected
- The need for better, deeper coverage analysis
- Idea: Upgrade our purely dynamic analysis based on a **call graph**:
  - oriented graph, which describes the mutual relationship of calls of individual program subroutines
  - **node** represents a specific subroutine
  - **edge** (f, g) indicates the call of g from within f

Proposed solution

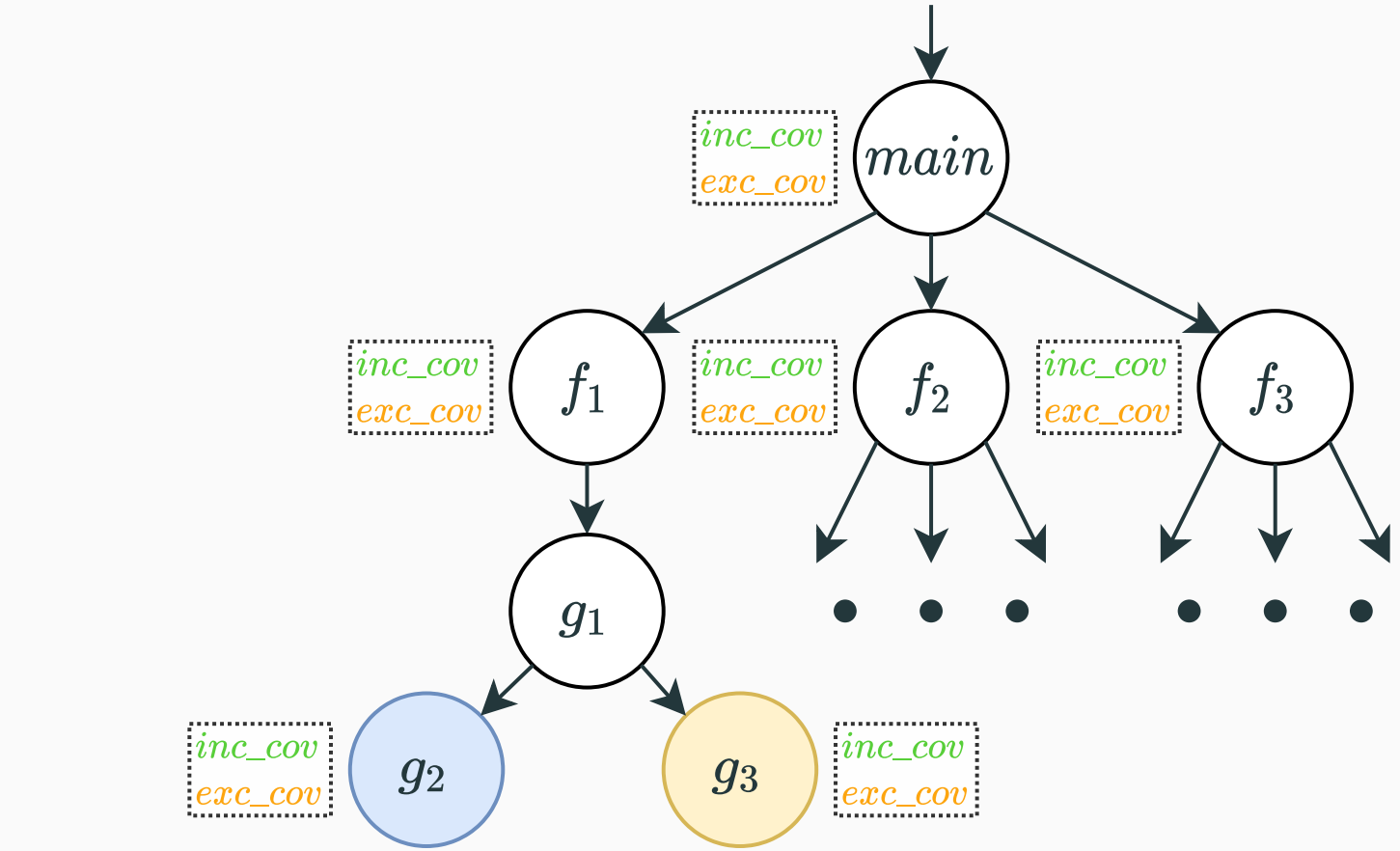
- call graph + coverage data = annotated call graph



- two types of subroutine coverage data:
  - *inclusive:* number of executed lines within the subroutine
  - *exclusive:* number of subroutine calls

How to construct coverage indicator

- Finding **all unique paths** from the root to the leaf node:



How to construct coverage indicator

- We define *inclusive* and *exclusive* coverage **of a path as sum** of the inclusive/exclusive coverage data of the path's subroutines:

$$inc\_cov(path) : \sum_{f \in path} inc\_cov(f)$$

$$exc\_cov(path) : \sum_{f \in path} exc\_cov(f)$$

- Our new **coverage indicator** will be a pair of coverage vectors we construct as follows:

$$inc\_vec = \Big(inc\_cov(path_1), inc\_cov(path_2), \dots \Big)$$

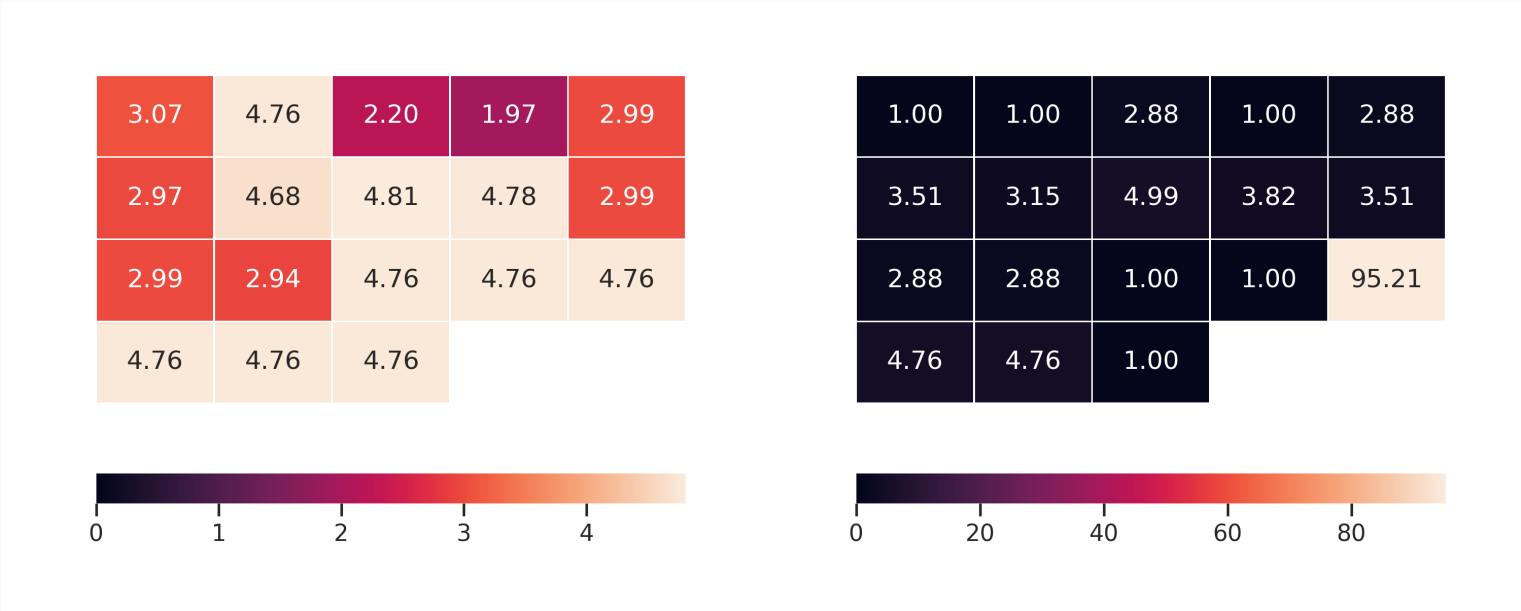
$$exc\_vec = \Big(exc\_cov(path_1), exc\_cov(path_2), \dots \Big)$$

Impact on the existing solution

- Modifying the Perun-fuzz:
  - How to determine the **baseline coverage**.
  - How to determine **interesting mutations** (whether it triggered sufficient coverage increase).
  - How to determine the **mutation score** if it caused a performance change.
- Resulting extension features:
  - More sophisticated and in-depth coverage analysis.
    - But also more computationally demanding.
  - Allows fuzzing of large projects.
  - More precise detection of detecting significant local changes in coverage.
  - Effective visualisation of new results about the paths dependency on the input.

Visualisation

- During fuzzing, we collect the information about **maximum** inclusive/exclusive **coverage increase** of the paths.
- The most influenced paths are included within fuzzing output.
- Max coverage ratio visualised using **heat maps**:



inclusive *exclusive*

Experimental Evaluation

Project	Old approach		New approach	
	cov ratio	hangs	cov ratio	hangs
(email+class) regex	5915.44	0	7.02	4
std::find, std::list	1.54	0	1.18	0
java hash function	3.58	0	3.56	0
stackoverflow regex	8.45	0	4.57	0
ubt	6.49	0	5.72	0

- Comparable results with previous approach.
- Can find timeouts (30 s) more quickly in programs with multiple performance hotspots.
- Better results on the larger projects, solid with the smaller ones.
- Deeper coverage analysis takes more time, but it is more precise.
- More evaluation is part of future work.

Summary

- Already done
  - ☑ Improve workload selection by deeper analysis of program:
    - ☑ Obtain the program call graph, find all unique paths and store in the proper representation.
    - ☑ Yield the necessary coverage data to annotate the call graph.
    - ☑ Remodel affected parts of fuzzing tool.
    - ☑ Provide the results of the analysis to the user in a suitable form.
- Next steps
  - ☐ Pull Request to Perun master branch.
  - ☐ Test and compare the fuzzer with the PerfFuzz.
  - ☐ Evaluate solution on real-world projects and potentially report new unique performance bugs.