PrograMet: Designing a Program Similarity Metric

Justin Wong

Columbia University

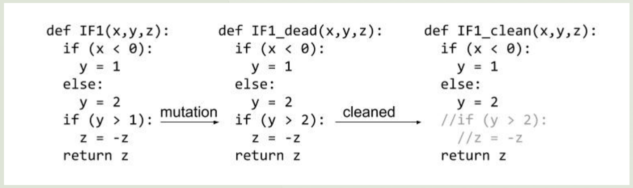
Abstract

We explore a new metric on programs, designed to capture program similarity beyond input/output behavior: PrograMet. In addition to taking hints from the syntactic structure of the program, PrograMet fully embraces execution traces as a valuable (albeit expensive) source of information. We evaluate PrograMet on prepared program samples including dead code introduced by unsatisfiable or tautological branching conditions. PrograMet desirably exhibits small distances from the original program when dead code is removed, while preserving large distances between that original program and similar code samples where both sides of the branch are reachable.

Motivation

How similar are two programs? This is a simple, yet open- ended question. A useful metric of similarity has powerful consequences from identifying code duplication to detecting malicious code and vulnerabilities. These tasks share an inherent challenge as small changes to the syntax of a program can lead to drastic differences in behavior. To make matters worse, often *good enough* programs suffice, and *interpretability* may be prioritized over behavior in practice. Thus, capturing similarity in a meaningful way between programs is intrinsically difficult as it requires the careful balancing of differences in how programs are written versus in how they behave.

Key Example



```
def IF1(x,y,z):
  if (x < 0):
    y = 1
  else:
    y = 2
  if (y > 1):
    z = -z
  return z
```

```
def IF1_dead(x,y,z):
  if (x < 0):
    y = 1
  else:
    y = 2
  if (y > 2):
    z = -z
  return z
```

```
def IF1_clean(x,y,z):
  if (x < 0):
    y = 1
  else:
    y = 2
  //if (y > 2):
    //z = -z
  return z
```

Consider the case of dead code, for example IF1_dead above:

- With no execution, IF1_dead seems extremely similar to the original IF1.
- Only using dynamic analysis, IF1_dead is identical to IF1_clean.
- Our ***goal*** is to design a metric that captures the intuition that IF1_dead is ***similar but not identical*** to IF1_clean.

Approach

Overview:

- We aim to combine static and dynamic approaches to construct PrograMet.
- We realize the static portion of the metric as the graph edit distance between control flow graphs.
- Meanwhile, the dynamic part (at present) tests all input values within a given domain and considers their "output similarity" and "trace similarity".
- The resulting metric results from the weighted sum of the static and dynamic contributions:

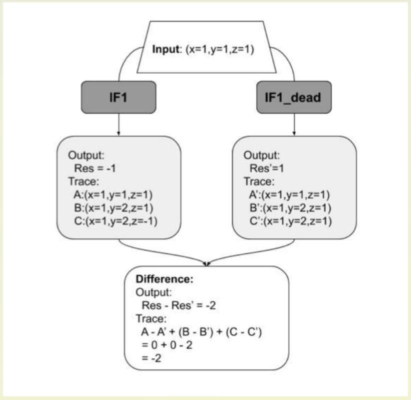
$$PM(P,Q) := \omega_S SM(P,Q) + \omega_D[OS(P,Q) + TS(P,Q)]$$

where P and Q are programs. SM is the static metric; OS the output similarity; and TS the trace similarity

Static Similarity (SM):

Currently we use graph edit distance of the control flow graph to quantify static differences in the programs

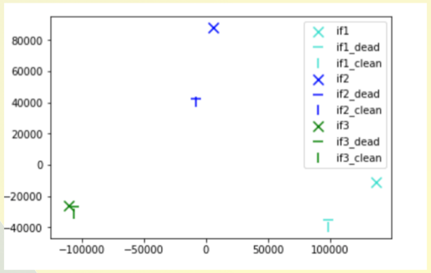
Dynamic Similarity:



As demonstrated in the figure, we compare execute the program and compare their traces and outputs to determine their similarity.

- **Output Similarity (OS):** OS evaluates output similarity by simply taking the canonical difference of the resulting value.
- **Trace Similarity (TS):** TS is summing the variable value differences for each recorded trace state.

Results



Embedding the full PrograMet metric using MDS:

- PrograMet separate different programs: IF1, IF2, and IF3.
- Further, the mutated "dead" versions are mapped similar to but not identical to the corresponding "clean" versions.

Conclusion and Future Directions

- PrograMet introduces initial steps towards balancing static and dynamic notions of similarity
- Results are promising and suggest that PrograMet can distinguish between two versions of code where a mutated if-condition leads to vastly different behavior.

As a whole, the primary goal of **future effort** is focused on coupling the static and dynamic components of program similarity.

- Currently, PrograMet treats static and dynamic similarity independently and then aggregates it.
- On approach is to overlay trace information over the control flow graph structure capturing properties such as reachability.

Acknowledgements

I'd like to thank Suman Jana and Andrew Blumberg for their mentorship and insightful conversations.