Using runtime behavioral signals to predict whether programs behave as intended

Claire Le Goues

(work largely performed by Deborah Katz, CMU graduate student)

TESTING/EXECUTION PROVIDES ONE TYPE OF EVIDENCE THAT REPAIR AND RECOVERY MECHANISM SHOULD BE *TRUSTED*.

Existing work provides important starting points for test *input* generation.

- Examples of test input generation strategies include:
  - Random.
  - Maximize various measures of code coverage.
  - Leverage previous knowledge of inputs that are often problematic.
- However, we still need expected *output*:
  - Some techniques use specifications or "gold standards", or global known properties (e.g., segmentation faults are bad.)
  - Generally, known unsolved problem.

GOAL: characterize normal program behavior such that we can generically identify abnormal behavior, in a program-specific, defect-independent way.

(solve the oracle problem)

INTUITION:

Dynamic runtime characteristics can be used to predict whether a program execution corresponds to expected behavior.

Intuition via example: Zune infinite loop

- On the last day of a leap year, enters an infinite loop on lines 3 through 16.

```
1  void zunebug(int days) {
2    int year = 1980;
3    while (days > 365) {
4      if (isLeapYear(year)){
5        if (days > 366) {
6          days -= 366;
7          year += 1;
8        }
9        else {
10       }
11     }
12     else {
13       days -= 365;
14       year += 1;
15     }
16   }
17   printf("current year is %d\n",
18   year);
19 }
```

http://news.bbc.co.uk/2/hi/technology/7806683.stm

Zune – expected behavior

- Calling with 730 produces expected behavior.
  - Represents 12/30/81.

```
1  void zunebug(int days) {
2    int year = 1980;
3    while (days > 365) {
4      if (isLeapYear(year)){
5        if (days > 366) {
6          days -= 366;
7          year += 1;
8        }
9        else {
10       }
11     }
12     else {
13       days -= 365;
14       year += 1;
15     }
16   }
17   printf("current year is %d\n",
18   year);
19 }
```

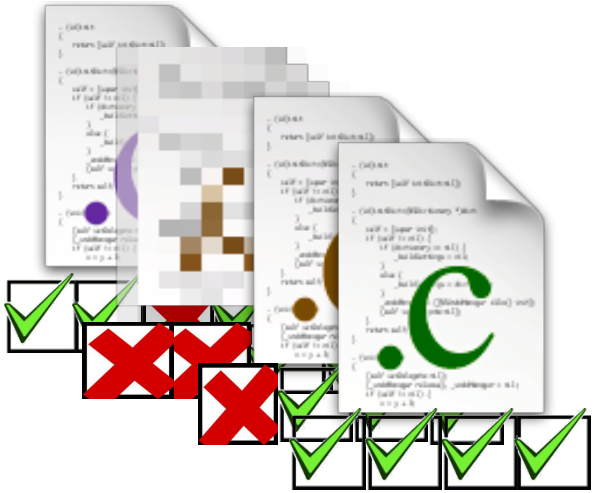
Zune – unexpected behavior

- Calling with 366 produces *unexpected* behavior.
  - Represents 12/31/80, last day of a leap year!
- Consider runtime signals that suggest that this behavior is abnormal:
  - Max program counter
  - Number of branches taken
  - Return value of isLeapYear function
  - Library code called
  - …

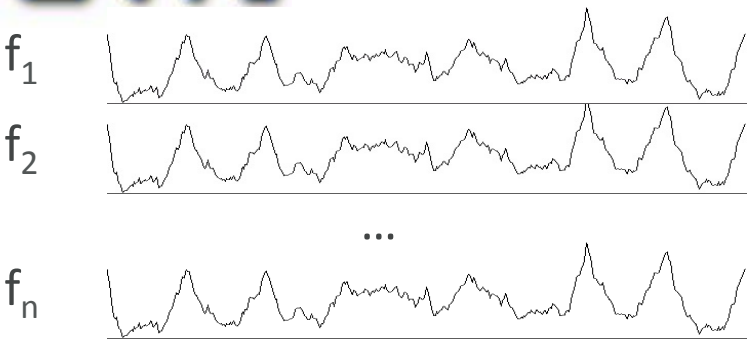
```
1  void zunebug(int days) {
2    int year = 1980;
3    while (days > 365) {
4      if (isLeapYear(year)){
5        if (days > 366) {
6          days -= 366;
7          year += 1;
8        }
9        else {
10       }
11     }
12     else {
13       days -= 365;
14       year += 1;
15     }
16   }
17   printf("current year is %d\n",
18   year);
19 }
```

Approach: use machine learning to relate program runtime characteristics to expected/unexpected behavior.

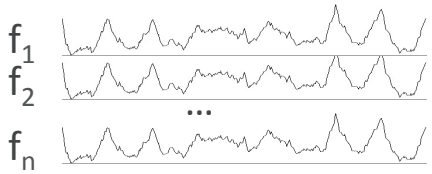
Input: Program variants, tests



Signal collection

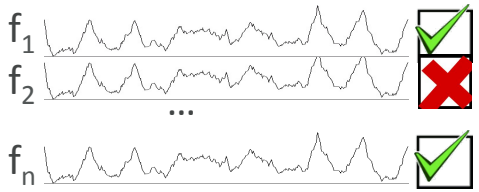


Model construction



(Unsupervised)

Model A



(Supervised)

Model B

(to be evaluated and deployed)

We used feature reduction to identify a core set of 15 predictive signals.

1. Fraction of statically-loaded instructions used.
2. Number of unique instructions executed.
3. Fraction of main executable instructions used.
4. Number of unique instructions used in main executable.
5. Fraction of routines executed.
6. Number of unique routines executed.
7. Mean of all addresses read.
8. Address of most frequent stack write.
9. Mean address of writes not on the stack or heap.
10. Mean distance between a read and the next write.
11. Mean distance between a write and the next write.
12. Number of instructions executed inside main executable.
13. Number of instructions executed in statically-loaded images but not main executable.
14. Fraction of instructions executed in main executable.
15. Number of instructions executed.

There are several scenarios in which this might be useful.

- Proof of concept: is this remotely reasonable?
  - Strategy: supervised machine learning on multiple versions of individual programs.
- What if we want to augment our existing tests?
  - Strategy: supervised machine learning based on a single version of a program.
- What if we have a dramatically under- or un-tested program?
  - Strategy: unsupervised machine learning over multiple versions of a program.
- What if we have no tests at all (but access to github)?
  - Strategy: supervised learning based on other programs and test suites.

		LOC	Versions	Tests
Siemens	printtokens	475	8	4130
	printtokens2	401	10	4115
	replace	512	33	5542
	schedule	292	10	2650
	schedule2	297	11	2710
	tcas	135	42	1608
	totinfo	346	24	1052
GenProg	deroff	1203	727	6
	gcd	22	141	6
	indent	4643	118	6
	svr-look	176	148	6
	ultrix-look	160	160	6
	uniq	562	159	6
	zune	20	150	24
CoreUtils	chgrp	249	27	63
	chmod	436	25	142
	chown	258	26	30
	cp	1003	30	264
	dd	1709	14	128
	df	1249	20	48

We need a dataset of programs with multiple versions and multiple test cases.

We anticipate actual deployment at the *method or node level* of systems of arbitrary size.

- Consider: ROS nodes

For initial investigation, focused on small programs that satisfied experimental needs, but can be mentally mapped to a smaller part of a larger system.

- Each set of programs presents interesting relevant characteristics.

Proof-of-concept

		Accuracy	Precision	Recall	F-Measure
Siemens	printtokens	0.76	0.99	0.77	0.87
	printtokens2	0.80	0.99	0.81	0.89
	replace	0.85	1.00	0.85	0.92
	schedule	0.74	0.99	0.74	0.85
	schedule2	0.81	1.00	0.81	0.90
	tcas	0.80	1.00	0.80	0.89
	totinfo	0.91	0.99	0.91	0.95
GenProg	deroff	0.89	0.96	0.85	0.90
	gcd	0.99	0.99	0.99	0.99
	indent	0.94	0.95	0.96	0.95
	svr-look	0.98	0.99	0.98	0.98
	ultrix-look	0.89	0.96	0.88	0.92
	uniq	1.00	1.00	1.00	1.00
	zune	0.91	0.89	0.89	0.89
CoreUtils	chgrp	0.80	0.98	0.80	0.88
	chmod	0.89	0.98	0.88	0.93
	chown	0.81	0.98	0.81	0.89
	cp	0.74	0.94	0.73	0.82
	dd	0.84	0.92	0.82	0.87
	df	0.61	0.51	0.55	0.53

Train and test within multiple versions of a single program.

- Training on balanced datasets, testing on unbalanced datasets.

Generally good performance across all metrics.

Precision especially high, which is good!

		Accuracy	Precision	Recall	F-Measure
Siemens	printtokens	0.73	0.99	0.73	0.84
	printtokens2	0.85	0.99	0.85	0.91
	replace	0.84	1.00	0.84	0.91
	schedule	0.81	0.99	0.81	0.89
	schedule2	0.80	1.00	0.80	0.89
	tcas	0.85	1.00	0.85	0.92
	totinfo	0.86	0.99	0.86	0.92
GenProg	deroff	0.61	0.89	0.53	0.67
	gcd	0.91	0.97	0.93	0.95
	indent	0.76	0.96	0.76	0.84
	svr-look	0.87	0.98	0.88	0.93
	ultrix-look	0.67	0.84	0.68	0.75
	uniq	0.67	0.99	0.66	0.79
	zune	0.86	0.83	0.83	0.83
CoreUtils	chgrp	0.75	0.96	0.75	0.84
	chmod	0.79	0.95	0.78	0.86
	chown	0.58	0.95	0.57	0.71
	cp	0.62	0.90	0.59	0.72
	dd	0.74	0.86	0.71	0.78
	df	0.52	0.39	0.45	0.41

Adding a new test case

Train and test on a single version, and test prediction accuracy for a new test input.

Generally good performance across all metrics.

Precision is still high.

		Accuracy	Precision	Recall	F-Measure
Siemens	printtokens	0.88	1.00	0.88	0.93
	printtokens2	0.90	0.98	0.92	0.89
	replace	0.92	0.99	0.92	0.96
	schedule	0.88	0.98	0.89	0.94
	schedule2	0.84	0.99	0.85	0.91
	tcas	0.95	0.99	0.95	0.97
	totinfo	0.91	0.97	0.94	0.95
GenProg	deroff	0.67	0.66	0.97	0.79
	gcd	0.57	0.47	1.00	0.64
	indent	0.75	0.73	0.99	0.84
	svr-look	0.80	0.79	0.98	0.87
	ultrix-look	0.78	0.79	0.98	0.87
	uniq	0.76	0.76	0.98	0.86
	zune	0.49	0.43	1.00	0.60
CoreUtils	chgrp	0.91	0.98	0.98	0.95
	chmod	0.77	0.82	0.92	0.87
	chown	0.90	0.98	0.97	0.95
	cp	0.79	0.81	0.96	0.88
	dd	0.62	0.64	0.92	0.75
	df	0.36	0.37	0.88	0.52

Unsupervised prediction within a program

Outlier detection, trained on multiple versions of a single program

More difficult than supervised learning, but still effective on many types of programs.

Recall is generally better than precision here.

		Accuracy	Precision	Recall	F-Measure
Siemens	printtokens	0.65	0.99	0.65	0.79
	printtokens2	0.67	0.96	0.68	0.80
	replace	0.34	0.98	0.34	0.50
	schedule	0.59	0.98	0.59	0.74
	schedule2	0.64	0.99	0.64	0.78
	tcas	0.36	0.95	0.37	0.53
	totinfo	0.60	0.94	0.61	0.74
GenProg	deroff	0.41	0.70	0.13	0.21
	gcd	0.67	0.81	0.21	0.33
	indent	0.34	0.50	0.38	0.43
	svr-look	0.81	0.96	0.77	0.85
	ultrix-look	0.78	0.79	0.95	0.86
	uniq	0.27	0.48	0.04	0.08
	zune	0.56	0.45	0.53	0.49
CoreUtils	chgrp	0.89	0.92	0.96	0.94
	chmod	0.70	0.88	0.82	0.82
	chown	0.86	0.93	0.92	0.92
	cp	0.52	0.81	0.53	0.64
	dd	0.48	0.63	0.45	0.52
	df	0.69	0.71	0.36	0.48

Generating tests for a new program based on other programs.

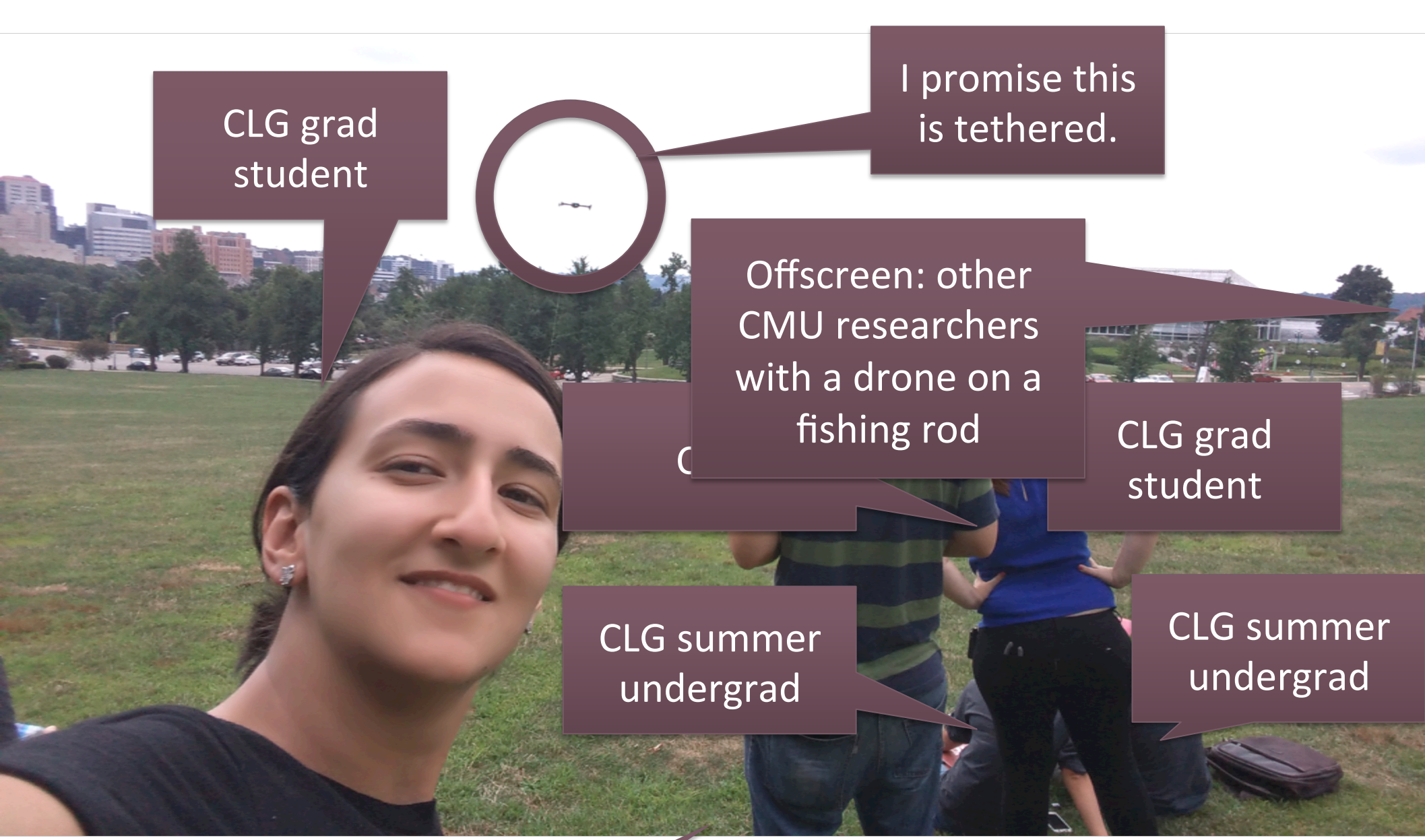
Train supervised models on other programs, use to predict correctness on held-out program.

Most difficult task for us, with wider variability in success.

However, results are fairly promising, especially since, again, precision is high.

Summary and future directions

- Collected dynamic signals about program execution and used machine learning to predict whether the executions passed or failed, achieving promising results for several applications.
- Present goal: analyze functions/modules within a larger system:
  - Tackling deployment to the quadcopters.


CLG grad student
I promise this is tethered.
Offscreen: other CMU researchers with a drone on a fishing rod
CLG grad student
CLG summer undergrad
CLG summer undergrad
this park is bigger than it looks

Summary and future directions

- Collected dynamic signals about program execution and used machine learning to predict whether the executions passed or failed, achieving promising results for several applications.
- Present goal: analyze functions/modules within a larger system:
  - Midway to deployment on the quadcopters.
  - Deploying to other autonomous systems in the context of robustness testing.
    - Collaboration with CMU's National Robotic's Engineering Center (NREC) funded by DoD's Test Resource Management Center (TRMC).
- Future directions:
  - Anomaly detection via learned dynamic structural system properties; Crossover with BRASS DARPA project (where we are analyzing collaborative grounded robots).
  - Test explicitly: How do models handle evolution? Our scenarios indicate that it works over multiple versions, but we need to study directly.

Other things I could talk about.

- Integrating models based on dynamic signals into robustness testing of safety-critical autonomous systems.
  - Some of this is ITAR, so I can speak in generalities.
  - (As a note: ROS also has many bugs.)
- Learning dynamic architectural properties of ROS systems.
- Program repair using semantic search.
  - Related to predicting, assuring, and measuring repair quality.