Lecture 22: Beyond Program Repair

(connecting repair to test-input generation)

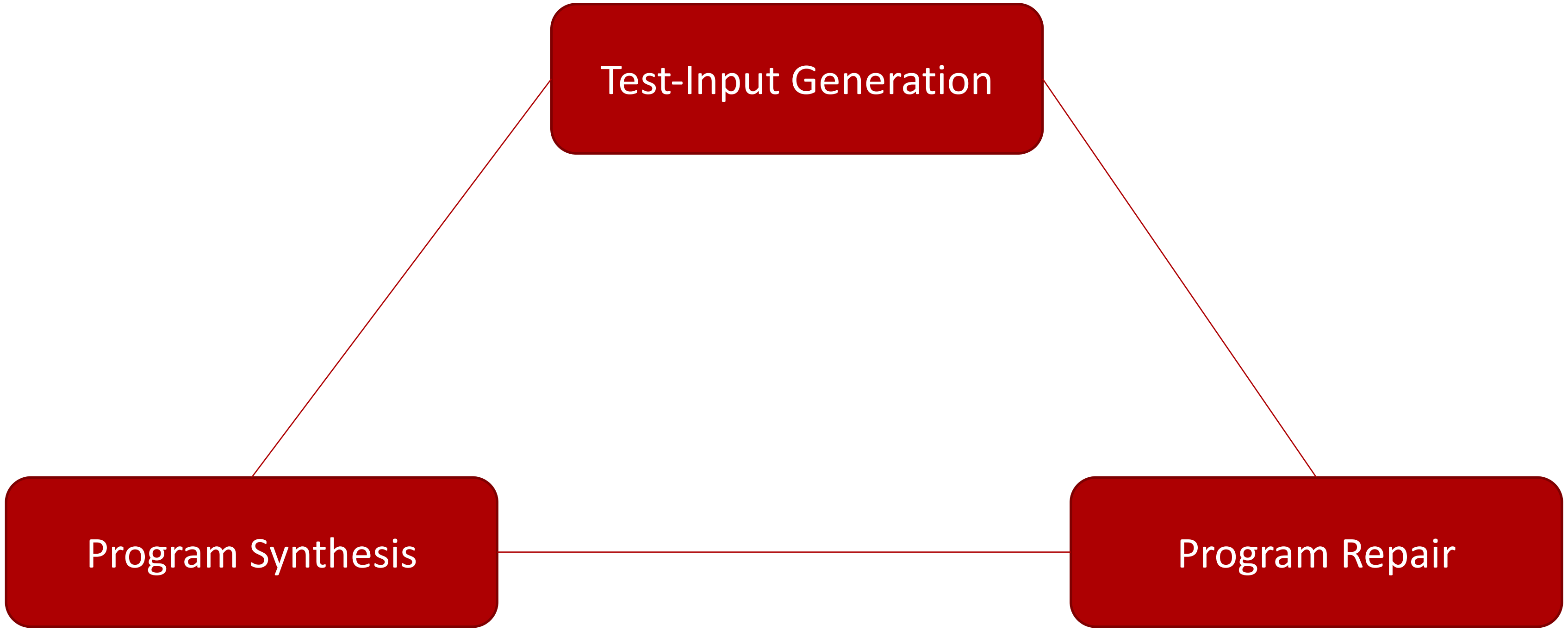
17-355/17-655/17-819: Program Analysis

Rohan Padhye and Jonathan Aldrich

April 22, 2021

* Course materials developed with Claire Le Goues

The Big Picture



Fundamentally, it's just a search problem

Fundamentally, it's just a search problem

- Bug Finding ≍ Reachability ≍ Static Analysis ≍ Test-Input Generation
- SMT Solvers for Test-Input Generation (Symbolic/Concolic Execution)
- Random Evolutionary Search for Test-Input Generation (Fuzzing)
- SMT Solvers for Program Synthesis (Semfix/Angelix)
- Random Evolutionary Search for Program Synthesis (GenProg)
- SMT Solvers & Random Search for Program Synthesis
- *Are all these problems equivalent???*
- *Can heuristics be reused?*

Adapting test-input generation tools to perform program repair

SYNTHESIS ≍ REACHABILITY

"Connecting Program Synthesis and Reachability: Automatic Program Repair using Test-Input Generation"
ThanhVu Nguyen, Westley Weimer, Deepak Kapur, and Stephanie Forrest. TACAS (2017).

Problem Formulation

- "Synthesis" $\asymp$ template-based program repair
  - Given a program P and failing test suite T, find program P′ such that T passes
  - Assume: P → P′ can be done by applying a *template* (think: sketch) at some known faulty location

- "Reachability" $\asymp$ test-input generation
  - Given a program P(x) and location L, find P(x) such that L is reached.

Motivating Example

```c
int is_upward(int in,int up,int down){
  int bias, r;
  if (in)
    bias = down; //fix: bias = up + 100
  else
    bias = up;
  if (bias > down)
    r = 1;
  else
    r = 0;
  return r;
}
```

	Inputs			**Output**		
Test	in	up	down	expected	observed	**Passed?**
1	1	0	100	0	0	✓
2	1	11	110	1	0	✗
3	0	100	50	1	1	✓
4	1	-20	60	1	0	✗
5	0	0	10	0	0	✓
6	0	0	-10	1	1	✓

Template-based Repair

Linear Templates:

$$\boxed{c_0} + \boxed{c_1} v_1 + \boxed{c_2} v_2$$

Template-based Repair

Linear Templates:

$$\boxed{c_0} + \boxed{c_1} v_1 + \boxed{c_2} v_2$$

Example:

$$\texttt{bias = } \boxed{c_0} \texttt{ +} \boxed{c_1} \texttt{*bias +} \boxed{c_2} \texttt{*in +} \boxed{c_3} \texttt{*up +} \boxed{c_4} \texttt{*down;}$$

Repair to Reachability

```
int c0,c1,c2,c3,c4; //global inputs

int is_upward_P(int in,int up,int
    down){
  int bias, r;
  if (in)
    bias =
     c0+c1*bias+c2*in+c3*up+c4*down;
  else
    bias = up;
  if (bias > down)
     r = 1;
  else
     r = 0;
```

Repair to Reachability

```
int c0,c1,c2,c3,c4; //global inputs

int is_upward_P(int in,int up,int
    down){
  int bias, r;
  if (in)
    bias =
     c0+c1*bias+c2*in+c3*up+c4*down;
  else
    bias = up;
  if (bias > down)
     r = 1;
  else
     r = 0;
  return r;
}

int main() {
  if(is_upward_P(1,0,100) == 0 &&
     is_upward_P(1,11,110) == 1 &&
     is_upward_P(0,100,50) == 1 &&
     is_upward_P(1,-20,60) == 1 &&
     is_upward_P(0,0,10) == 0 &&
     is_upward_P(0,0,-10) == 1){
    [L]
  }
  return 0;
}
```

Reachability to Repair

```
//global inputs
int x, y;

int P(){
  if (2 * x == y)
    if (x > y + 10)
      [L]

  return 0;
}
```

Reachability to Repair

```
//global inputs
int x, y;

int P(){
  if (2 * x == y)
    if (x > y + 10)
      [L]

  return 0;
}
```

```
int P_Q() {
  if (2*[x] == [y])
    if([x] > [y]+10)
      //loc L in P
      raise
         REACHED;

  return 0;
}
```

Reachability to Repair

```
//global inputs
int x, y;

int P(){
  if (2 * x == y)
    if (x > y + 10)
      [L]

  return 0;
}
```

```
int P_Q() {
  if (2*[x] == [y])
    if([x] > [y]+10)
      //loc L in P
      raise
         REACHED;

  return 0;
}
```

```
int main_Q() {
  //synthesize x, y
  int x = c_x;
  int y = c_y;
  try
    P_Q();
  catch (REACHED)
    return 1;

  return 0;
}
```

CETI (Correcting Errors using Test Inputs)

- Benchmark program: tcas (written in C; 1608 tests and 41 faults)
- Fault localization: Tarantula (top-80 locations)
- Front-end: CIL
- Templates: modify constants and operators
- CETI transforms templates to reachability instances
- CETI uses KLEE to exhaustively search test program space

"Connecting Program Synthesis and Reachability: Automatic Program Repair using Test-Input Generation"
ThanhVu Nguyen, Westley Weimer, Deepak Kapur, and Stephanie Forrest. TACAS (2017).

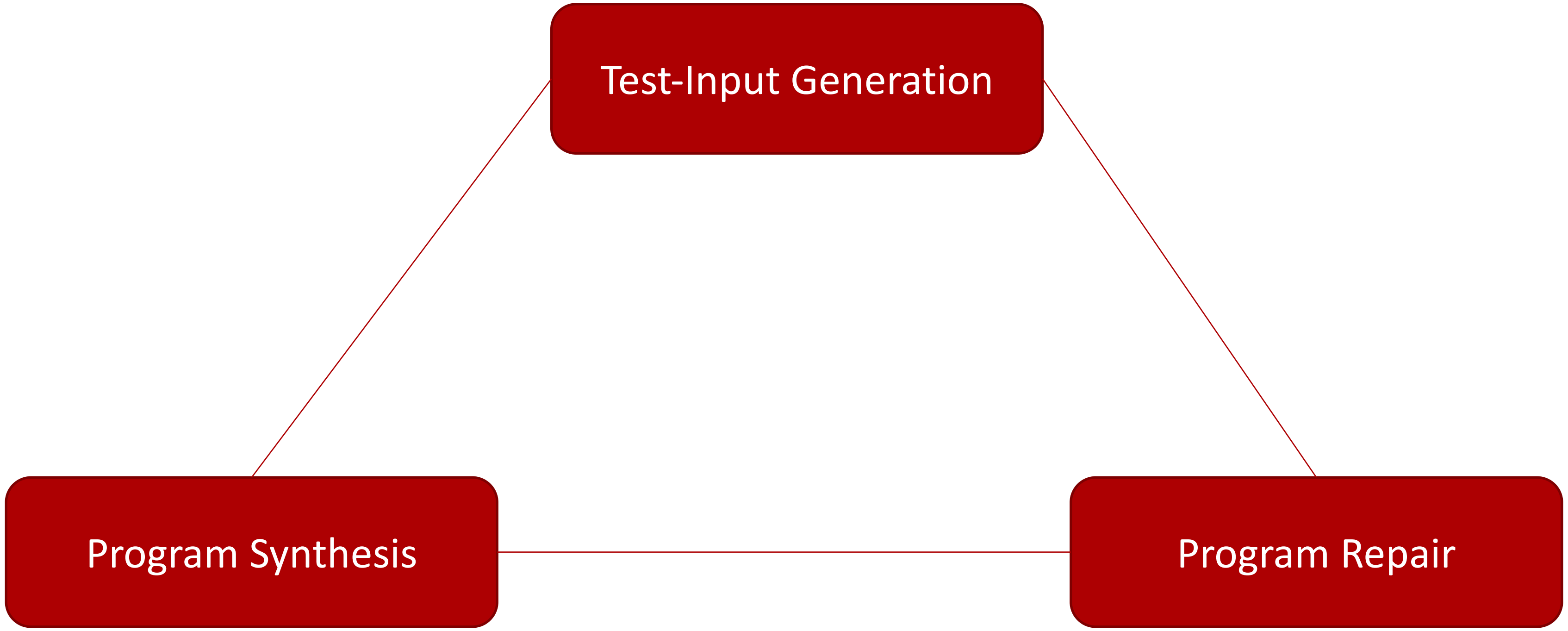
Evaluation on **tcas**

- GenProg repairs 11 of 41 defects
- Semfix repairs 34 of 41 defects
- **CETI** repairs 26 of 41 defects

Table 1. Repair Results for 41 Tcas Defects

	Bug Type	R-Progs	T(s)	Repair?		Bug Type	R-Progs	T(s)	Repair?
v1	incorrect op	6143	21	✓	v22	missing code	5553	175	–
v2	missing code	6993	27	✓	v23	missing code	5824	164	–
v3	incorrect op	8006	18	✓	v24	missing code	6050	231	–
v4	incorrect op	5900	27	✓	v25	incorrect op	5983	19	✓
v5	missing code	8440	394	–	v26	missing code	8004	195	–
v6	incorrect op	5872	19	✓	v27	missing code	8440	270	–
v7	incorrect const	7302	18	✓	v28	incorrect op	9072	11	✓
v8	incorrect const	6013	19	✓	v29	missing code	6914	195	–
v9	incorrect op	5938	24	✓	v30	missing code	6533	170	–
v10	incorrect op	7154	18	✓	v31	multiple	4302	16	✓
v11	multiple	6308	123	–	v32	multiple	4493	17	✓
v12	incorrect op	8442	25	✓	v33	multiple	9070	224	–
v13	incorrect const	7845	21	✓	v34	incorrect op	8442	75	✓
v14	incorrect const	1252	22	✓	v35	multiple	9070	184	–
v15	multiple	7760	258	–	v36	incorrect const	6334	10	✓
v16	incorrect const	5470	19	✓	v37	missing code	7523	174	–
v17	incorrect const	7302	12	✓	v38	missing code	7685	209	–
v18	incorrect const	7383	18	✓	v39	incorrect op	5983	20	✓
v19	incorrect const	6920	19	✓	v40	missing code	7364	136	–
v20	incorrect op	5938	19	✓	v41	missing code	5899	29	✓
v21	missing code	5939	31	✓					

The Big Picture



Fundamentally, it's just a search problem

Recap: Challenges in Test-Input Generation

- Oracles
    - What is a bug? Crash? Silent overflow? Infinite loop? Race condition? Undefined behavior? How do we know when we have found a bug?
- Debugging
    - Reproducibility
    - **Crash triaging**
    - Input minimization
- Roadblocks
    - Dependencies in binary inputs (e.g. length of chunks, indexes into tables – see PNG)
    - Inputs with complex syntax and semantics (e.g. XML, JSON, C++)
    - Stateful applications

Crash Triaging

- Given two crashing inputs x1 and x2, do they trigger the same bug?
- *Very* difficult to answer in practice
- Herustics: bug(x1) = bug(x2) only if.... (consider pros/cons of each)
  - exitcode(x1) = exitcode(x2) // or exception or error msg
  - coverage(x1) = coverage(x2)
  - stacktrace(x1) = stacktrace(x2)
  - newcoverage(x1, old) = newcoverage(x2, old) // AFL
  - **fix(x1) = fix(x2)**

What if we could actually tell if they have the same fix???

Semantic Crash Bucketing

Rijnard van Tonder
School of Computer Science
Carnegie Mellon University
USA
rvt@cs.cmu.edu

John Kotheimer
Heinz College
Carnegie Mellon University
USA
john.kotheimer@alumni.cmu.edu

Claire Le Goues
School of Computer Science
Carnegie Mellon University
USA
clegoues@cs.cmu.edu

ABSTRACT

Precise crash triage is important for automated dynamic testing tools, like fuzzers. At scale, fuzzers produce millions of crashing inputs. Fuzzers use heuristics, like stack hashes, to cut down on duplicate bug reports. These heuristics are fast, but often imprecise: even after deduplication, hundreds of uniquely reported crashes can still correspond to the same bug. Remaining crashes must be inspected manually, incurring considerable effort. In this paper we present Semantic Crash Bucketing, a generic method for precise crash bucketing using program transformation. Semantic Crash Bucketing maps crashing inputs to unique bugs as a function of changing a program (i.e., a semantic delta). We observe that a real bug fix precisely identifies crashes belonging to the same bug. Our insight is to *approximate* real bug fixes with lightweight program transformation to obtain the same level of precision. Our approach uses (a) patch templates and (b) semantic feedback from the program to automatically generate and apply approximate fixes for general bug classes. Our evaluation shows that approximate fixes are competitive with using true fixes for crash bucketing, and significantly outperforms built-in deduplication techniques for three state of the art fuzzers.

1 INTRODUCTION

The advent of large scale fuzzing services, such as Google's OSS-Fuzz [1, 45] and Microsoft's fuzzing service [9], attest to the effectiveness of automatic bug finding tools. When operating at scale, accurately identifying unique bugs is critical for (a) reducing time-consuming manual debugging efforts [14, 41], (b) characterizing the effectiveness of automated bug-finding tools [12, 14, 37, 42, 48], and (c) ranking interesting crashing test cases [14]. However, one outstanding challenge in effectively deploying automated fuzzing techniques is accurately identifying unique bugs during crash triage. Fuzzers often generate thousands of crashing inputs that ultimately correspond to the same bug [14], and the sheer number of crashing inputs preclude manual inspection. This is a hard problem, and an area of active research [17].

Automated crash triage techniques seek to approximately *bucket* multiple crashing (but ultimately equivalent) inputs [14, 17, 37, 41], to reduce the number of redundant bug reports an engineer must inspect by hand. At a high level, automated testing tools like fuzzers and symbolic executors typically use tool-specific, heuristic bucketing strategies. Both research and industry standard triage techniques have known limitations [17, 42]. Techniques may assume

Key Idea:
Use templates to generate (approximate) fixes for crashes to cluster into equivalence classes

Problem Formulation

- Given a set of n crashing inputs $c_1, c_2, c_3, \dots c_n$.
  - Program *P* crashes when executed with any input c_i.
- Let $B = \{b_1, b_2, \dots b_m\}$ be a set of bugs in the program ($n \geq m$)
  - Each bug b_i is represented as a bucket of unique crashes $\{c_{b_{i1}} \dots\}$
- Let $T_i: P \rightarrow P$ be a transformation of program P that fixes bug i.

Ideal Bucketing

- Given a set of n crashing inputs $c_1, c_2, c_3, \dots c_n$.
  - Program *P* crashes when executed with any input c_i.
- Let $B = \{b_1, b_2, \dots b_m\}$ be a set of bugs in the program $(n \geq m)$
  - Each bug b_i is represented as a bucket of unique crashes $\{c_{b_{i1}} \dots\}$
- Let $T_i: P \rightarrow P$ be a transformation of program P that fixes bug i.

$$
\begin{aligned}
&\forall\, b_i \in B, \\
&\forall\, b_j \in B \setminus b_i \text{ s.t.} \\
&\qquad \forall\, c_i \in b_i, \langle T_i(P),\ c_i \rangle \not\rightsquigarrow \mathit{crash} \\
&\qquad \forall\, c_j \in b_j, \langle T_i(P),\ c_j \rangle \rightsquigarrow \mathit{crash}
\end{aligned}
$$

Imprecise Bucketing

- Given a set of n crashing inputs $c_1, c_2, c_3, \ldots c_n$.
  - Program *P* crashes when executed with any input c_i.
- Let $B = \{b_1, b_2, \ldots b_m\}$ be a set of bugs in the program $(n \geq m)$
  - Each bug b_i is represented as a bucket of unique crashes $\{c_{b_{i1}} \ldots\}$
- Let $T_i : P \to P$ be a transformation of program P that fixes bug i.

$$
\begin{aligned}
&\exists\, b_i \in B, \\
&\exists\, b_j \in B \setminus b_i \ \texttt{s.t.} \\
&\qquad \forall\, c \in b_i, \langle T_i(P),\ c\rangle \not\rightsquigarrow \mathit{crash} \\
&\qquad \exists\, c_{\mathrm{dup}} \in b_j, \langle T_i(P),\ c_{\mathrm{dup}}\rangle \not\rightsquigarrow \mathit{crash}
\end{aligned}
$$

Approximate Fixes with Templates

Null Dereferences

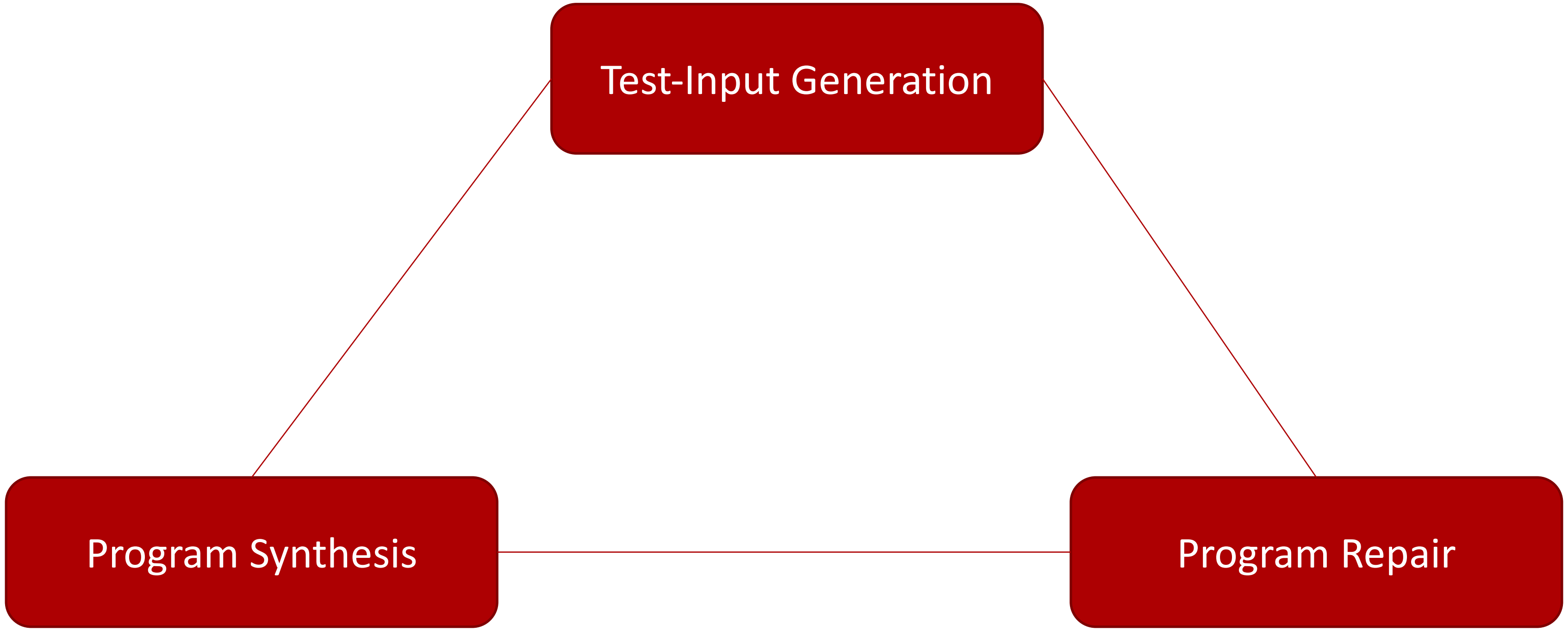
```
if (%%%PVAR%%% == null) {
  exit(101);
}
```

Use GDB to locate crash point and work backwards to find a good place to apply template

Buffer Overflows – reduce length

```
// Modify a possible overflowing memcpy call
size_t angelic_length = 1;
memcpy(%%%DST%%%,%%%SRC%%%,angelic_length);
```

The Big Picture



Fundamentally, it's just a search problem

PROJECTS DISCUSSION