Automatic Rootcausing for Program Equivalence Failures in Binaries

Shuvendu K. Lahiri (Microsoft Research)

Rohit Sinha (UC Berkeley)

Chris Hawblitzel (Microsoft Research)

Large scale program equivalence checking

- Compiler translation validation
  - [Pnueli et al. TACAS'98, Necula PLDI'00, …]
- Cross-version verification
  - [Godlin & Strichman DAC'09, Lahiri et al. CAV'12,…]
- Verifying student solutions against reference implementations
  - [Singh et al. PLDI'13]

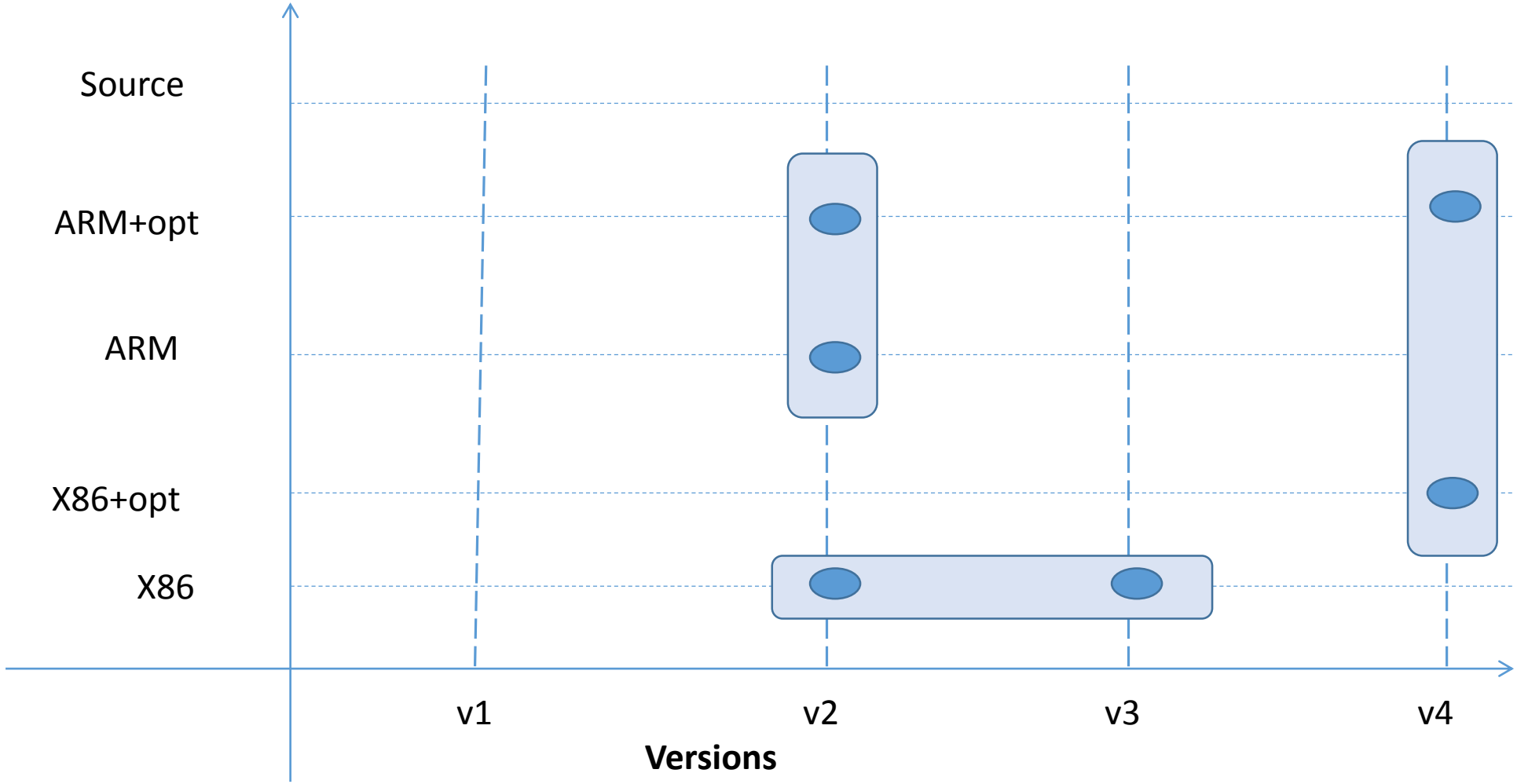
Motivation: Rootcausing equivalence failures

- Provide effective feedback to users of the tool
  - Dealing with **thousands of equivalence failures**

- Compiler translation validation
  - Same alarm manifests in hundreds of test programs

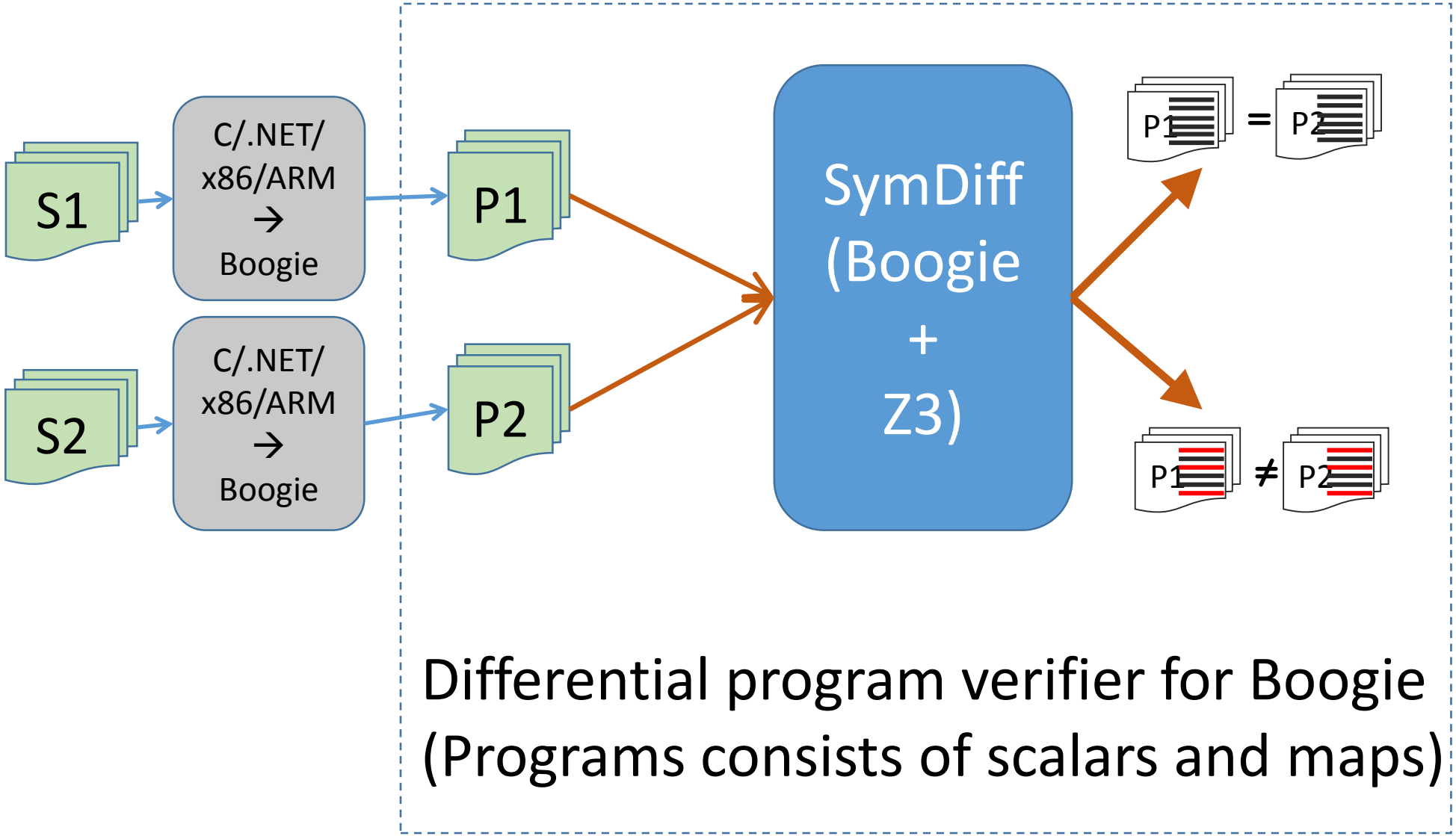
- Comparing student attempts
  - Many students often make similar mistakes

Application: Compiler validation



- Validating the CLR .NET compiler [Hawblitzel, Lahiri et al. FSE'13]
  - SymDiff to compare two assembly/binary programs

Verification flow



Bucketization of equivalence failures

- Almost 500,000 test C# methods pushed through the tool
  - Was applied by CLR test team for several months, found real bugs
  - Even 1-2% false alarm ➔ ~several thousand warnings

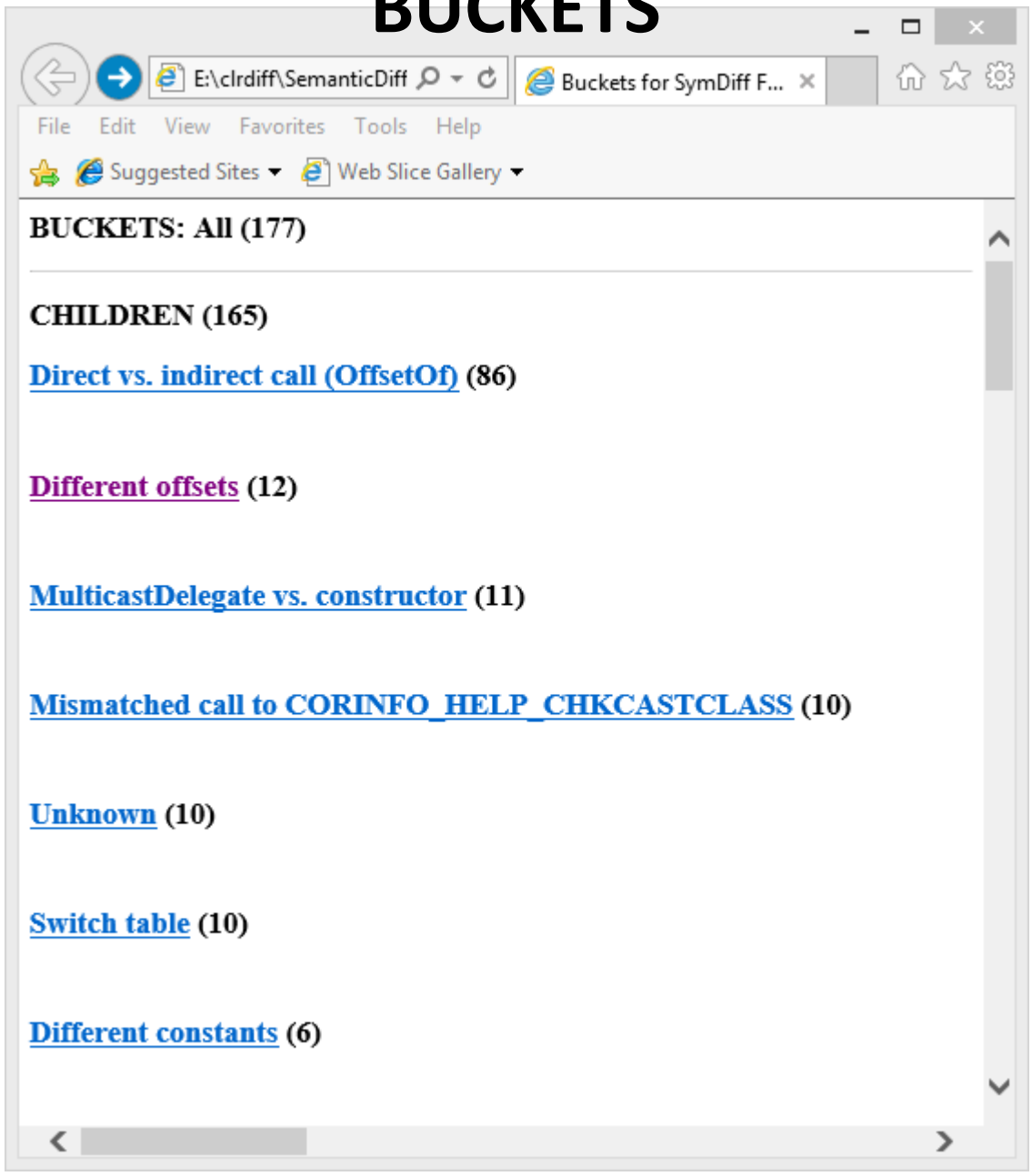
- Main ask from users
  - Need to group failures into a small number of buckets

- Each bucket captures one source of equivalence failure
  - Different manifestations of the same bug
  - Different manifestations of same false alarm

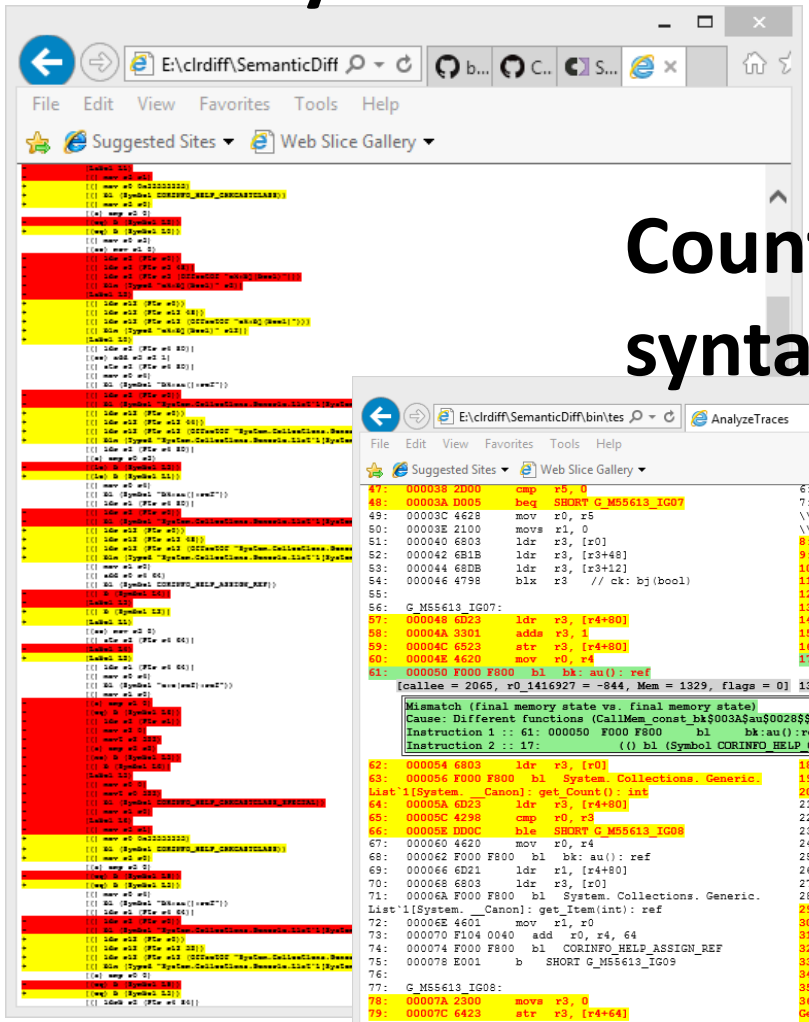
False alarms

- Often due to **modular checking** and **missing domain-knowledge**
  - Concrete addresses
    - 0x004fe208 vs 0x003dd484
  - Different memory layout by the compiler
    - Field stored in two different offsets: [eax + 4] vs. [eax + 32]
  - Aliasing assumptions known only to the compiler
    - Store to address x does not modify address y
  - Side effects of procedures
    - A procedure call does not modify certain heap locations
    - Purity
  - ….

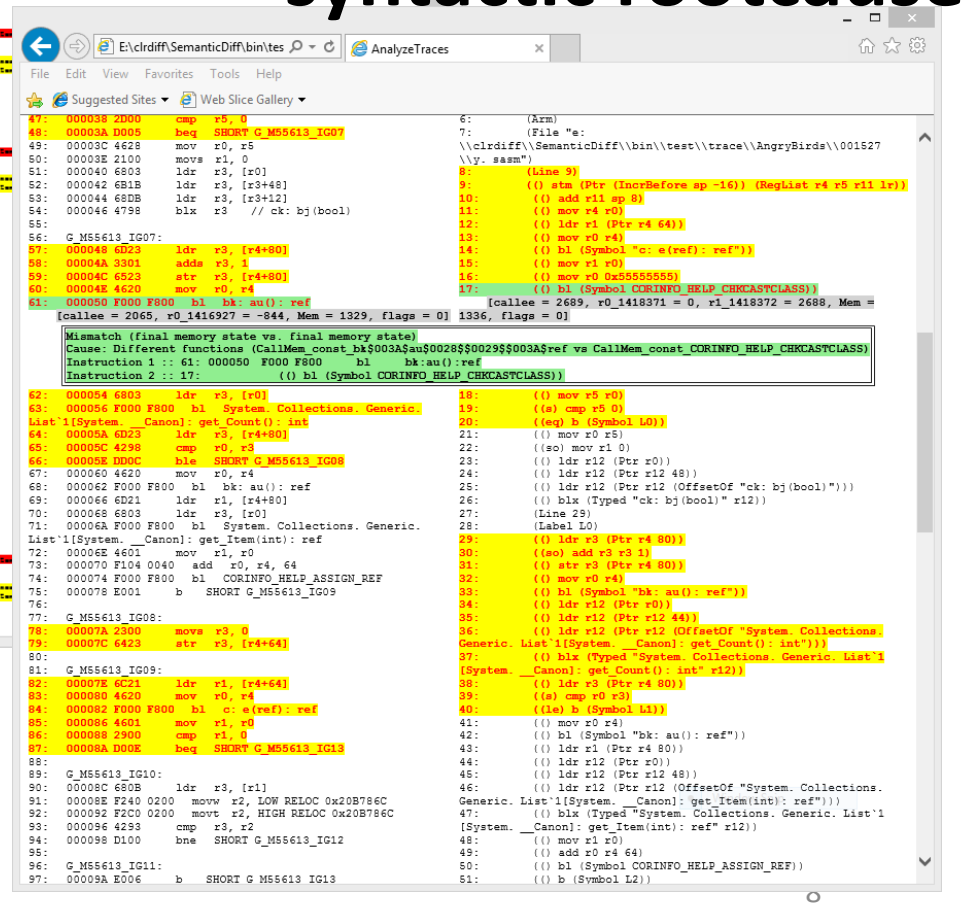
BUCKETS



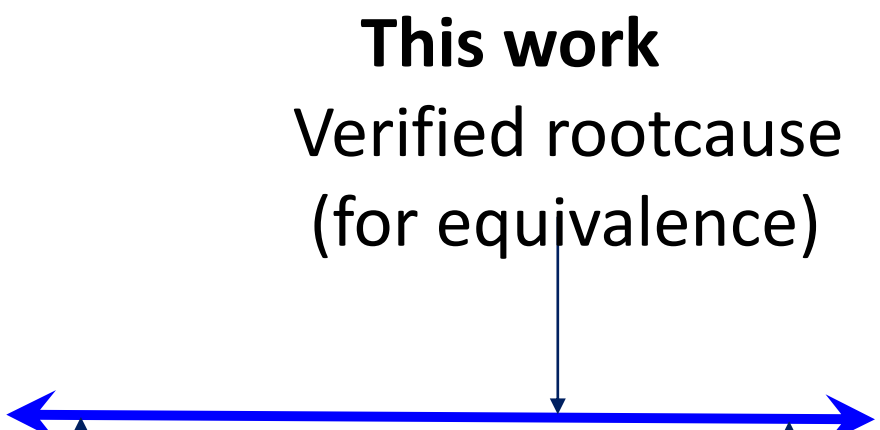
Syntax diffs



Counterexamples + syntactic rootcause



Prior works



Fault localization
(Jose et al. PLDI'11, ..)

- Provides a program slice (often large for eq checking of binaries)
- Little guarantee

Program Repair
(Nguyen et al. ICSE'13, Singh et al. PLDI'13, …

- Need to **repair** the program
- Needs a template of repair
- Scalability

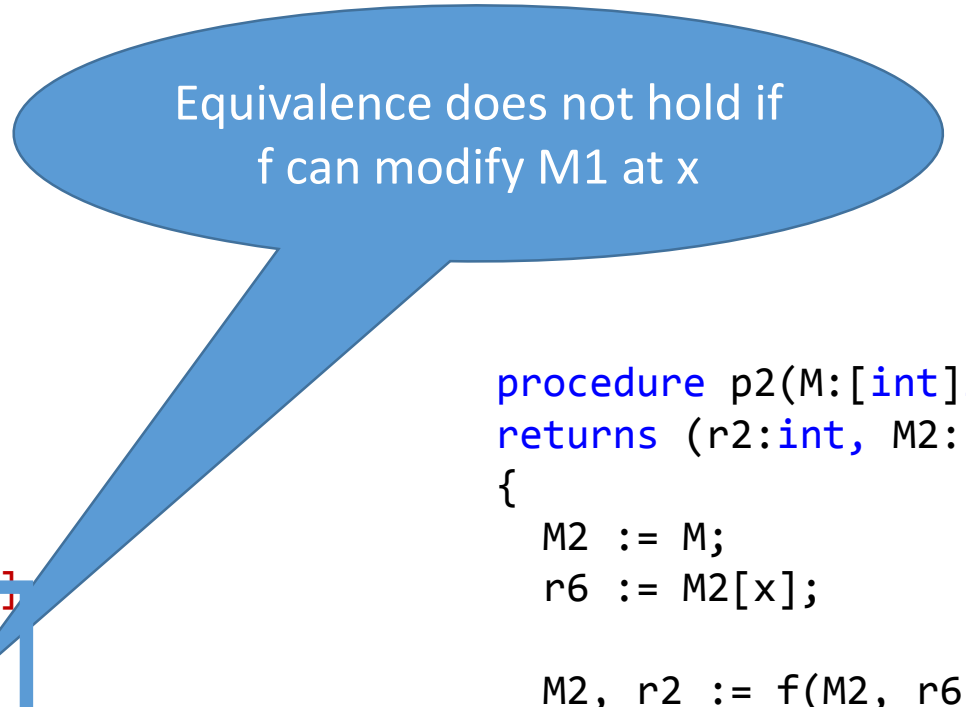
Wish list for rootcausing

- **Formalize** a valid rootcause at Boogie level (and thus can verify)
  - Points out the **first pair of instructions** where programs diverge
- **Automatic**
  - Providing templates difficult for failures due to modeling imprecision
- Can express **domain knowledge at the Boogie level**
  - Ideas can be agnostic to the source programs
  - Reusable for other programs (x86/ARM/x64/C/Java)

This talk

- Formalization of rootcause for equivalence failures
  - For structurally similar programs
- Dealing with the need for multiple fixes
- Implementation
  - Optimizations (MAXSAT, Binary search)
- Evaluation

Example




```
procedure p1(M:[int]int, x:int)
returns (r1:int, M1:[int]int)
{
  M1 := M;
  r5 := M1[x];              //ld r5, [x]

  M1, r1 := f(M1, r5);  //call f
  r1 := r5;             //mov r1, r5
  M1, r1 := f(M1, r1);  //call g
  return;
}
```

```
procedure p2(M:[int]int, x:int)
returns (r2:int, M2:[int]int)
{
  M2 := M;
  r6 := M2[x];              //ld r6, [x]

  M2, r2 := f(M2, r6);  //call f
  r2 := M2[x];              //ld r2, [x]
  M2, r2 := g(M2, r2);  //call g
  return;
}
```

Optimized

≠

Insight: Fix p2 by using values computed by p1

Optimization: Remove redundant load

Rootcause definition

- For procedures P1, P2 and a counterexample, a *fix* ***(L,R)*** is a pair of assignments *L: x := e* (in P1), and *R: y := e'* (in P2) such that

1. replacing e' (in P2) with *the value of* e at L (in P1), makes P1 and P2 equivalent, and
2. (L, R) is the *earliest* such pair satisfying (1)

- Note that a "fix" does not repair P2
  - Not the same as replacing expression e' with e

Example

Captures the value from p1

Replaces the value in p2

```
const r5@0: int;
procedure p1(M:[int]int, x:int)
returns (r1:int, M1:[int]int)
{
  M1 := M;
  r5 := M1[x];            //ld r5, [x]

  assume r5@0 == r5;

  M1, r1 := f(M1, r5);    //call f
  r1 := r5;               // mov r1, r5
  M1, r1 := f(M1, r1);    //call g
  return;
}
```

Optimized

=

```
procedure p2(M:[int]int, x:int)
returns (r2:int, M2:[int]int)
{
  M2 := M;
  r6 := M2[x];            //ld r6, [x]

  M2, r2 := f(M2, r6);    //call f
  r2 := M2[x];            //ld r2, [x]

  r2 := r5@0;

  M2, r2 := g(M2, r2);    //call g
  return;
}
```

Extra load

Optimization: Remove redundant load

Instrumentation

- Left program

  For each scalar assignment instruction $l : \mathtt{x} := \mathtt{e}$ with a label $l \in L$, we transform it to:

  $$l : \mathtt{x} := \mathtt{e}; \mathtt{assume}(\theta@l = \mathtt{x})$$

- Right program

  For each assignment instruction $r : \mathtt{x} := \mathtt{e}$ with a label $r \in R$, we transform it to:

  $$r : \mathtt{x} := \mathtt{e};\ \mathtt{x} := \gamma_r ?\ \theta@r :\ \mathtt{x};\ \mathtt{assume} \bigwedge_{l \in L} (\beta_r^l \Rightarrow \mathtt{x} = \theta@l)$$

Benefit of the formulation

- Naturally captures debugging equivalence failures
  - Provides a program pair that helps with debugging
  - Useful for bucketization
- Automatic (when such a pair exists)
- Do not need to solve the (more difficult) *repair* problem
- Exploits the *similarity of computations* on both sides

Challenge: multiple fixes

- Two cases
  - Single fix along multiple paths
  - Multiple fixes along single path

Single fix along multiple paths

- Fix *only one path* in the **left** program
  - Formally: rootcause verified for all inputs that exercise the counterexample path in the left program (weaker guarantee than all inputs)
  - May take the **right** program along a different control flow path

- Exploits the structural similarity of the two programs
  - Unlike previous work that treats one program as a black-box [Singh et al. PLDI‘13], hence need to **repair** the **entire program**

Multiple fixes along a path

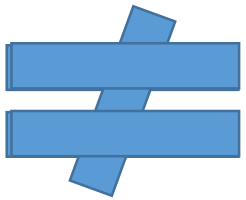
- Encode domain knowledge as additional preprocessing
- Weaken the equivalence check

1. Fix *intermediate synchronization* points
   - E.g. State of the heap has to be identical after procedure calls
   - Weaken the final equivalence check with intermediate equivalence that failed
2. Constrain callee summaries

Can be expressed as preprocessing of Boogie programs

Constraining Callees : Weaker Fix

```
procedure p1(M:[int]int, x:int)
returns (r1:int, M1:[int]int)
{
  M1 := M; r1 := M1[x];
  assume M1@0 == M1;
  M1, r1 := getLength(M1, r1);
  assume M1 == M1@0; //no side-effect on heap
  assume r1@0 == r1;
  if (r1 > 0) {
    M1, r1 := writeToFile(M1, r1);
  }
  ...
}
```



```
procedure p2(M:[int]int, x:int)
returns (r2:int, M2:[int]int)
{
  M2 := M; r2 := M2[x];
  r2 := M2[r2 + 8];
  r2 := r1@0;
  if (r2 > 0) {
    M2, r2 := writeToFile(M2, r2);
  }
  ...
}
```

Domain knowledge: If a callee (e.g. getLength) appears in only one of the programs, treat it as side-effect-free

MAXSAT-based optimization

- If P1 and P2 have n assignments each, our naïve algorithm explores $O(n^2)$ candidate fixes.
- Only a small set of candidate pairs actually fixes P2
  - How do we prune away the rest? (difficult to get concrete runs due to uninterpreted functions)
- Pose it as a MAXSAT problem for any assignment in P2

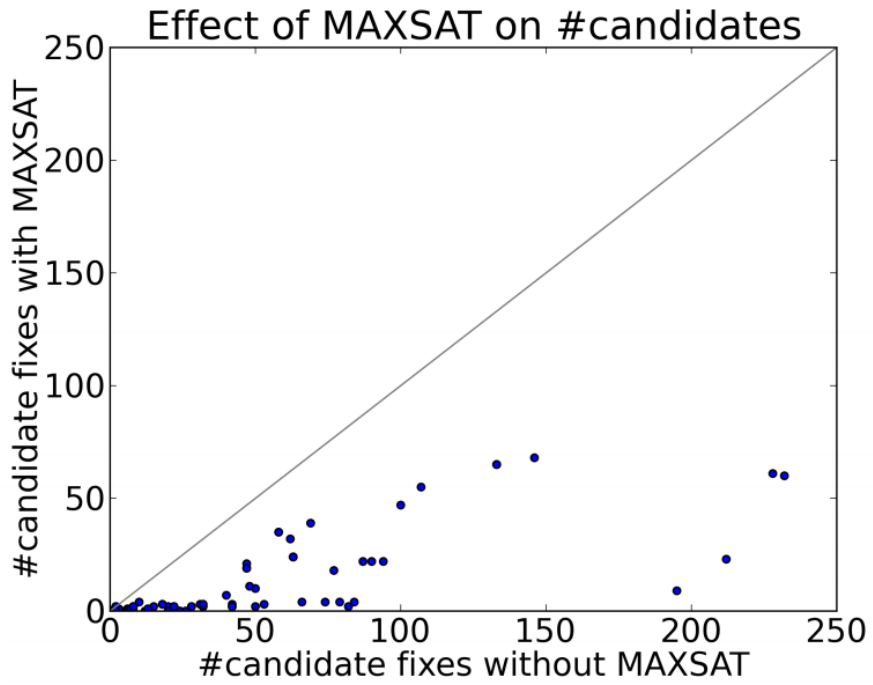
For each assignment instruction $r : \mathtt{x} := \mathtt{e}$ with a label $r \in R$, we transform it to:

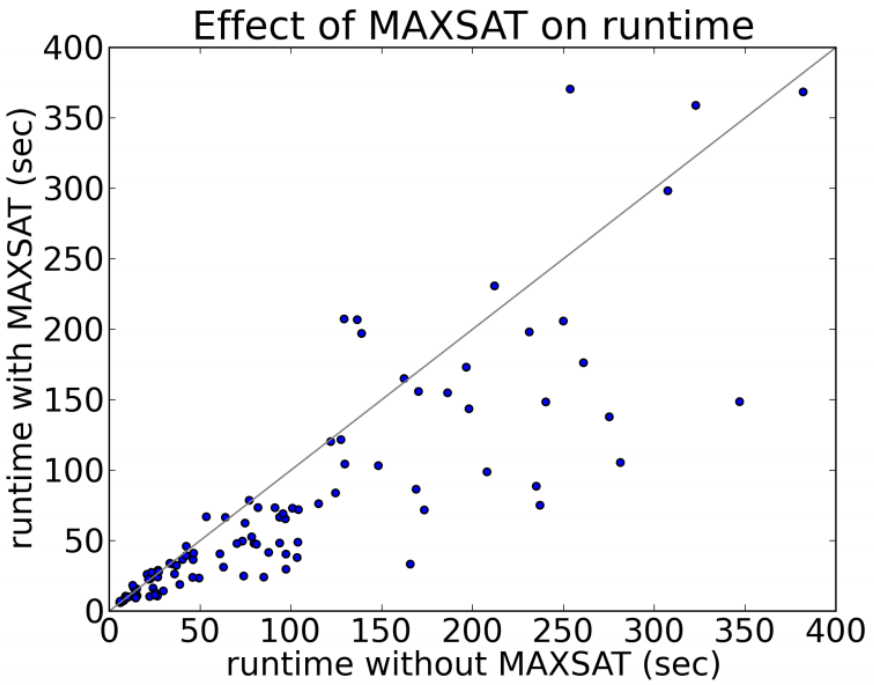
$$r : \mathtt{x} := \mathtt{e};\ \mathtt{x} := \gamma_r ?\ \theta@r : \mathtt{x};\ \mathtt{assume} \bigwedge_{l \in L} (\beta_r^l \Rightarrow \mathtt{x} = \theta@l)$$

What is the maximal subset of conjuncts that satisfies
P1 != P2

Effect of MAXSAT optimization

- Average 49% improvement in runtime, and 4x reduction in the number of candidates





Evaluation

- A representative sample of benchmarks from earlier work

- **JIT vs. compiled binaries**
  - Average of 165 assembly instructions (1242 Boogie statements) per procedure
  - Found rootcause in 34/46 benchmarks (74% of cases)

- **x86 vs. Optimized x86**
  - Average of 68 assembly instructions (510 Boogie statements) per procedure
  - Found rootcause in 12/15 small benchmarks (80% of cases)

Conclusion

- Natural formulation of verified rootcause for equivalence failures of similar programs
  - Automatic
  - Can be extended to several cases requiring multiple fixes
  - Rootcause integrated into SymDiff Codeplex
- Future Directions:
  - Combine with CEGIS (multiple fixes)
  - Application to automatic grading of student submissions in MOOCs

http://research.microsoft.com/symdiff

Questions