Differential and cross-version program verification

Shuvendu Lahiri

Research in Software Engineering (RiSE),

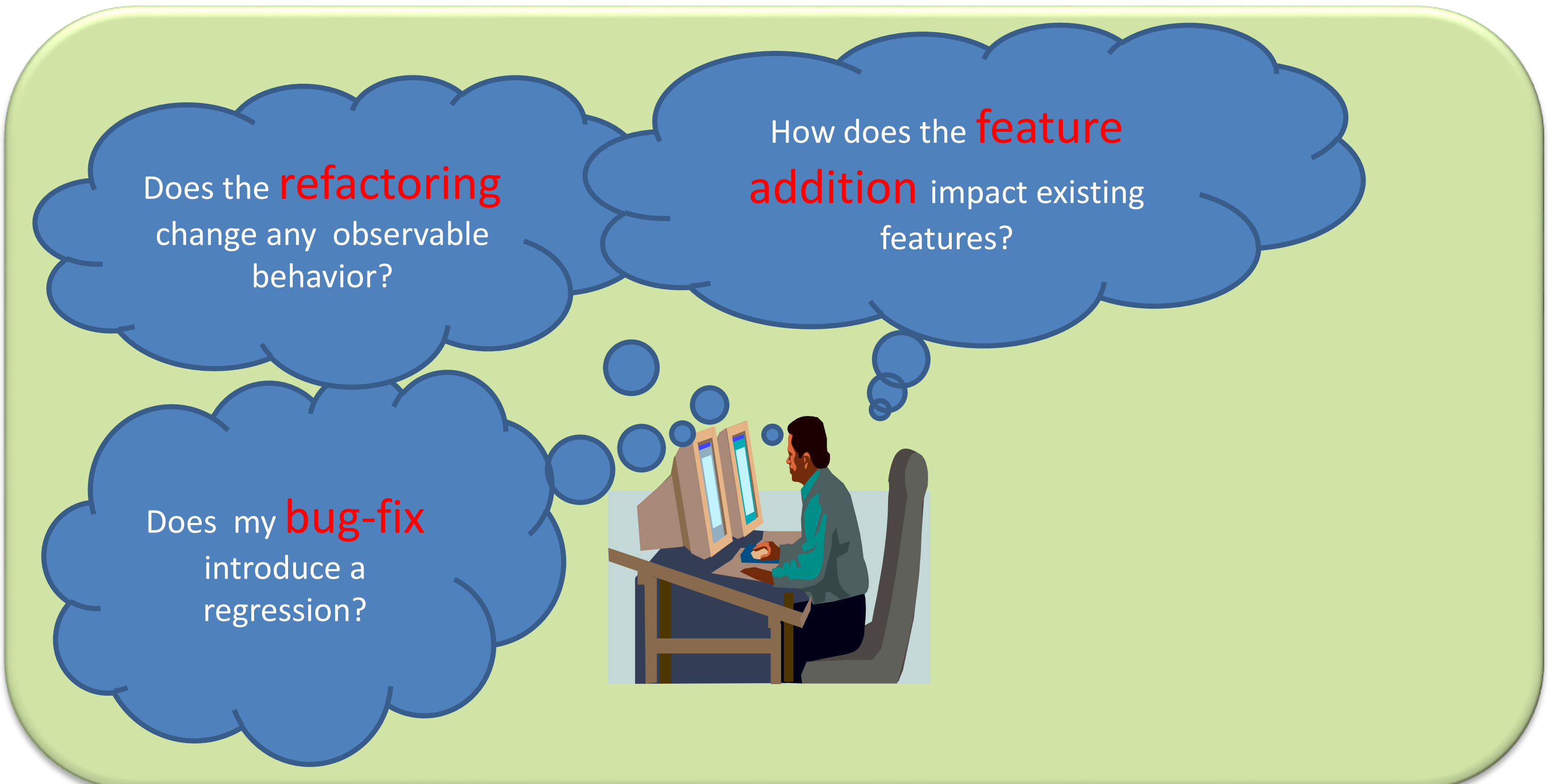
Microsoft Research,

Redmond, WA USA

Halmstad Summer School on Testing (HSST 2015)

Software evolution

– Programmers spend a large fraction of their time ensuring (read *praying*) **compatibility** after changes



Changes

- Bug fixes
- Feature addition (response to a new event)
- Refactoring
- Optimizations
- Approximations (tradeoff accuracy for efficiency)
- …

Main question

- Can we preserve the *quality* of a software product as it evolves over time?
- Currently, testing and code review are the only tools in ensuring this
  – Useful, but has its limitations (simple changes take long time to checkin, no assurance on change coverage)
- How do we leverage and extend program verifiers towards differential reasoning?
  – Relatively new research direction

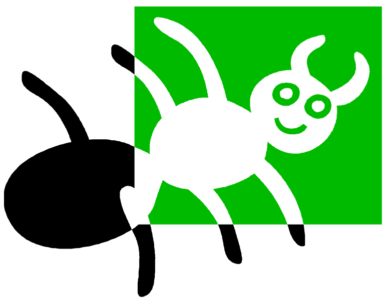
Outline

- Motivation
- SymDiff: A differential program verifier
  - Program verification background
  - Differential specifications
  - Differential program verification
- SymDiff: Applications
- Other applications of differential reasoning for existing verifiers
  - Verification modulo versions, Interleaved bugs
- Other works in differential cross-version program analysis
- Works in differential analysis of independent implementations

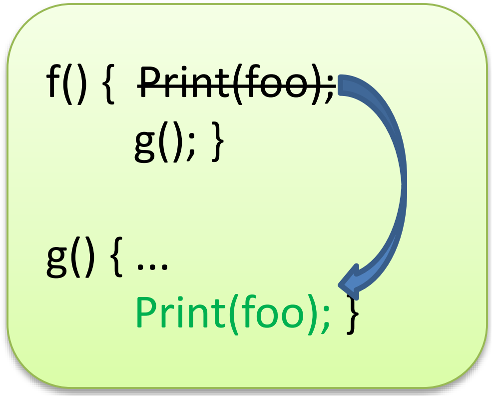
What will you learn

- Some flavor of program verification using SMT solvers
- Modeling of imperative programs for verification
- Formalizing differential specifications
- Practical automated, differential verification in **SymDiff**
- Applying differential verifier to improve existing verifiers
- Applications of differential analysis (cross version and independent implementations)
- Try out examples in **SymDiff** (Windows drop currently)

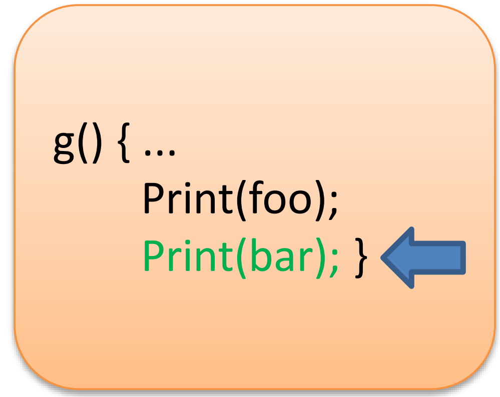
Compatibility: applications



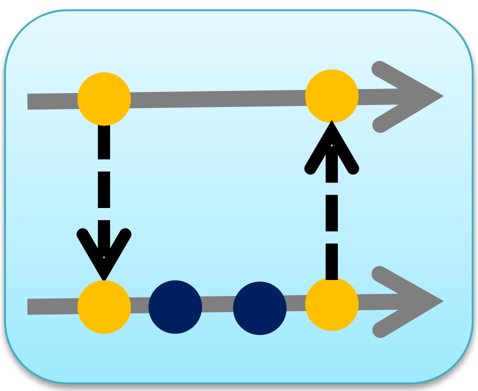
Bug fixes



Refactoring

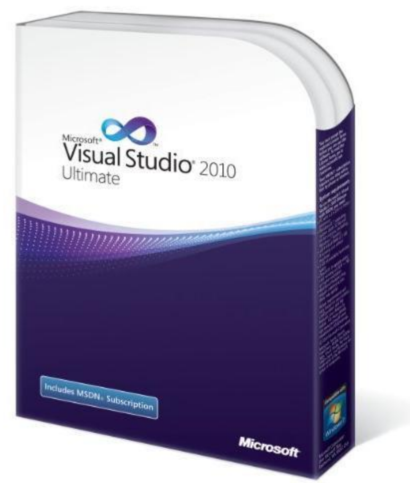


New features



Version Control

Library API changes



Compilers

Equivalence checking in hardware vs software

Hardware

- One of commercial success story of formal verification
  http://en.wikipedia.org/wiki/Formal_equivalence_checking
- Routinely applied after timing optimizations
- Commercial products
- Almost considered a solved research problem

Software

- Most changes are not semantics preserving
- Explaining equivalence failure needs users to understand the low-level modeling of programs (e.g. in the presence of heap)

Motivation

- Ensure code changes **preserve** quality
  - Help developers gain greater confidence for relatively simple changes through program verification
- Cost effectiveness of program verification
  - Only success stories in last several decades in the hands of a few expert users, or domain-specific properties (e.g. SLAM/SDV)
    - Need for specification
    - Scalability
    - Need for complex program-specific invariants
    - Environment models

What is SymDiff?

A framework to

– Leverage and extend program verification for *differential verification*

Source code
http://symdiff.codeplex.com/

Install direction
http://symdiff.codeplex.com/documentation

Papers etc.
http://research.microsoft.com/symdiff

Outline

- ✓ Motivation
- SymDiff: A differential program verifier
  - Program verification background
  - Differential specifications
  - Differential program verification
- SymDiff: Applications
- Other applications of differential reasoning for existing verifiers
  - Verification modulo versions, Interleaved bugs
- Other works in differential cross-version program analysis
- Works in differential analysis of independent implementations

Demo

- Equivalence
- DAC and relative verification

Program verification: background

Program verification

- A simple imperative language (**Boogie**)
  - Syntax
  - Modeling heap
- Specifications
  - How to write the property to be checked
- Verification
  - How to check that a given property holds
- Invariant Inference
  - How to automatically generate intermediate facts

Boogie

- Simple intermediate verification language
  - [Barnett et al. FMCO'05]
- Commands
  - x := E //assignments
  - havoc x //change x to an arbitrary value
  - assert E //if E holds, skip; otherwise, go wrong
  - assume E // if E holds, skip; otherwise, block
  - S ; T //execute S, then T
  - goto L1, L2, … Ln //non-deterministic jump to labels
  - call x,y := Foo(e1,e2,..) //procedure call

Boogie (contd.)

- Two types of expressions
  - Scalars (bool, int, ref, ..)
  - Arrays ([int]int, [ref]ref, …)
- Array expression sugar for SMT array theory
  - x[i] := y → x := upd(x, i, y)
  - y := x[i] → y := sel(x,i)
- old(e): Value of an expression at entry to the procedure

Procedure specifications

- Each procedure has a specification (default **true**)
- Procedure calls can be replaced with their specifications

```
procedure Foo();
requires  pre;
ensures  post;
modifies x,y,z;
```

requires pre; → precondition

ensures post; → postcondition

modifies x,y,z; → set of variables possibly modified in Foo

```
call Foo();
```

→

```
assert pre;
havoc x, y, z;
assume post;
```

Modeling imperative features

- Popular languages (e.g. C) support other features
  - Pointers
  - Structures/classes
  - Address-of operations
  - ..
- Various front-ends from such languages to Boogie
  - C (HAVOC/SMACK/VCC/..)
  - JAVA (Joogie/..)
  - C# (BCT)

Translating Heap

  - [Condit, Hackett, Lahiri, Qadeer POPL'09]
- HAVOC memory model
  - A pointer is represented as an integer (`int`)
  - One heap map per scalar/pointer structure field and pointer type
  - struct A { int f; A* g;} x;

```
Mem_f_A : [int]int
Mem_g_A : [int]int
Mem_A: [int]int
```

- Simple example
  - C code

```
x->f = 1;
```

  - Boogie

```
Mem_f_A[x + Offset(f,A)] := 1;
```

C → Boogie

```c
typedef struct {
 int g[10]; int f;} A;

A *create() {
   int a;

   A *d = (A*)
   malloc(sizeof(A));
   init(d->g, 10, &a);

  d->f = a;
  d->g[1] = 2;

   return d;
}
```

```
function g_A(:int) : int {u + 0}
function f_A(u:int): int {u + 40}

procedure create() returns d:int{
    var @a: int;
    call @a := malloc(4);
    call d := malloc(44);

    call init(g_DATA(d),10, @a);

    Mem_f_A[f_A(d)] := Mem_INT[@a];
    Mem_g_A[g_A(d) + 1*4]:=2;
    free(@a);
    return;
}
```

(Modular) verification problem

- Given a program P
  - A list of procedures p1, p2, …
  - Each procedure has assert , requires, ensures
- Verify that each procedure satisfies its specifications/contracts (assuming the contracts of other procedures)

Verification using VC + SMT

  - Assume loops are tail-recursive procedures (for the rest of this talk)
- Verification condition (VC) generation
  - A quadratic encoding of each procedure p into a logical formula VC(p)
    - If VC(p) is **valid** *then* p satisfies its contracts
- Check the validity of each of VC(p) using an SMT solver (e.g. Z3, YICES, CVC4, ..)
  - Efficient solvers for Boolean combination over various theories (arithmetic, arrays, quantifiers, …)
  - [http://smtlib.cs.uiowa.edu/]

Quick summary of VC generation

- [Barnett&Leino FMCO'05, Godefroid & Lahiri LASER'11]

- High-level steps
  - Replace procedure calls with their specifications
    - call F(e) → {assert pre_F; havoc x_F; assume post_F;}
  - Eliminate assignments
    - Perform static single assignment (SSA) for variables
    - Replace an assignment x_i := E with assume x_i == E
  - Perform *weakest precondition* for statements in each basic block
  - Replace goto statements with block equations

VC Generation

A { start: $x := 1$; goto l_1;

B { l_1: $x := x + 1$; goto l_2, l_3;

C { l_2: assume $x == 0$;
$x := x + 2$;
goto l_4;

D { l_3: assume $x \neq 0$;
$x := x + 3$;
goto l_4;

E { l_4: assert $x == 5$

VC Generation

A { start: assume $x_0 == 1$; goto l_1;

B { l_1: assume $x_1 == x_0 + 1$; goto l_2, l_3;

C { l_2: assume $x_1 == 0$;
assume $x_2 == x_1 + 2$;
assume $x_4 == x_2$; goto l_4;

D { l_3: assume $x_1 \neq 0$;
assume $x_3 == x_1 + 3$;
assume $x_4 == x_3$; goto l_4;

E { l_4: assert $x_4 == 5$

$$A_{ok} \Leftrightarrow (x_0 == 1 \Rightarrow B_{ok})$$

$$\wedge$$

$$B_{ok} \Leftrightarrow (x_0 == 1 \Rightarrow C_{ok} \wedge D_{ok})$$

$$\wedge$$

$$\wedge$$

$$C_{ok} \Leftrightarrow (x_1 == 0 \Rightarrow$$
$$(x_2 == x_1 + 2 \Rightarrow$$
$$(x_4 == x_2 \Rightarrow E_{ok})))$$

$$\wedge$$

$$D_{ok} \Leftrightarrow (x_1 \neq 0 \Rightarrow$$

$$\wedge$$

$$(x_2 == x_1 + 3 \Rightarrow$$
$$(x_4 == x_3 \Rightarrow E_{ok})))$$

$$E_{ok} \Leftrightarrow (x_4 == 5 \wedge \text{true})$$

$$\Rightarrow \quad A_{ok}$$

VC Generation

Formula over Arithmetic, Equality, and Boolean connectives

Can be solved by a SMT solver

$$
\begin{aligned}
& A_{ok} \Leftrightarrow (x_0 == 1 \Rightarrow B_{ok}) && \wedge \\
& B_{ok} \Leftrightarrow (x_0 == 1 \Rightarrow C_{ok} \wedge D_{ok}) && \wedge \\
& C_{ok} \Leftrightarrow (x_1 == 0 \Rightarrow && \wedge \\
& \qquad (x_2 == x_1 + 2 \Rightarrow \\
& \qquad (x_4 == x_2 \Rightarrow E_{ok}))) && \wedge \\
& D_{ok} \Leftrightarrow (x_1 \neq 0 \Rightarrow && \wedge \\
& \qquad (x_2 == x_1 + 3 \Rightarrow \\
& \qquad (x_4 == x_3 \Rightarrow E_{ok}))) \\
& E_{ok} \Leftrightarrow (x_4 == 5 \wedge \text{true}) \\
& \Rightarrow A_{ok}
\end{aligned}
$$

Invariant inference

- Challenge: user needs to write down every pre/post condition for **modular** verification to succeed
- Infer "program facts" that are true
  - Missing loop invariants, procedure pre/post conditions
- Can be eager or lazy (property-driven)
  - Eager (*abstract interpretation* [Cousot&Cousot POPL'77])
  - Lazy (*counterexample guided abstraction refinement* (CEGAR) [Clarke et al. CAV'00])

Boogie demo

- Input C program
- Intermediate Boogie program

Outline

- ✓ Motivation
- SymDiff: A differential program verifier
  - ➢ Program verification background
  - – Differential specifications
  - – Differential program verification
- SymDiff: Applications
- Other applications of differential reasoning for existing verifiers
  - – Verification modulo versions, Interleaved bugs
- Other works in differential cross-version program analysis
- Works in differential analysis of independent implementations

SymDiff

- How do we leverage program verifiers for differential verification
  - How do we specify differential properties
  - How do we check the properties
  - How do we infer intermediate invariants

Differential specifications

(Partial) Equivalence

- Procedures p and p’ are *partially equivalent* if
  - For all input states i, if p terminates in o and p’ terminates in o’, then o == o’

- Notes
  - Verifying equivalence is undecidable for programs with loops and unbounded counters
  - Procedure may not-terminate (loops), and may have multiple outputs for an input (non-determinism)

Specifying equivalence

- Construct a **product procedure** EQ_p_p'

```
procedure EQ_p_p'(i, i'): (o,o') {
    call o :=  p(i);         //modifies g
    call o' := p'(i');       //modifies g'
}
```

- Write a postcondition
  - ensures (i == i' && old(g) == old(g') ==> o == o')
  - ensures (i == i' && old(g) == old(g') ==> g == g')

- Caveats
  - Note that we are comparing entire arrays for equality (good and bad)!
  - Specification is easy, but verification often require more than equivalence

Factorial

```
f1(n): returns r {
  if (n == 0) {
    return 1;
  } else {
    return n * f1(n - 1);
  }
}
main(n) : r {r := f1(n);}
```

```
f2(n, a) : returns r {
  if (n == 0) {
    return a;
  } else {
    return f2(n - 1, a * n);
  }
}
main(n) : r {r := f2(n,1);}
```

```
procedure EQ_main_main'(n, n'): (r, r');
ensures (n == n' ==> r == r')
```

Equivalence too strong

- Most software changes are not equivalence preserving
  - Bug fixes, feature additions, adding logging, ..
- Need more relaxed specifications (failure points to likely regressions)
  - Generic specifications
    - Differential assertion checking
    - Control-flow equivalence
  - Manual specifications

Differential assertion checking (DAC)

  - [Lahiri et al. FSE'13, Joshi, Lahiri, Lal POPL'12]
- Correctness → Relative correctness
  - Check that an input that does not fail assertion in p does not fail an assertion in p'
- How to specify
  - Construct EQ_p_p' procedure
  - Replace assert A → ok := ok && A;
  - Write a postcondition

  ensures (i == i' && old(g) == old(g') ==> (ok ==> ok'))

- Note: asymmetric check

Relative Correctness (fails)

```
void strcopy_correct
(char* dst, char*src,
int size)
{
  int i = 0;
  for(;i<size-1 &&
       *src; i++)
    *dst++ = *src++;
  *dst = 0;
}
```

```
void strcopy_buggy
(char* dst, char*src,
int size)
{
  int i = 0;
  for(;*src &&
       i<size-1; i++)
    *dst++ = *src++;
  *dst = 0;
}
```

CEX: size=0, src =0, dst= some valid location

Relative Correctness (Passes)

```
void strcopy_buggy
(char* dst, char*src,
int size)
{
  int i=0;
 for(;*src &&
       i<size-1; i++)
    *dst++ = *src++;
  *dst = 0;
}
```

```
void strcopy_correct
(char* dst, char*src, int
size)
{
int i=0;
  for(;i<size-1 &&
       *src; i++)
    *dst++ = *src++;
  *dtmp = 0;
}
```

- No need to constrain the inputs
- Verifying absolute correctness needs preconditions and complex program-specific loop invariants

Mutual summaries

  - [Hawblitzel, Kawaguchi, Lahiri, Rebelo CADE’13]
- General form of differential specification
  - Captures EQ and DAC specifications
- Create a procedure similar to EQ_p_p’
  - We name it as MS_check_p_p’ as the body of the procedure is more complex (later)

Mutual summaries

```
void F1(int x1){
  if(x1 < 100){
      g1 := g1 + x1;
      F1(x1 + 1);
  }
}
```

```
void F2(int x2){
  if(x2 < 100){
      g2 := g2 + 2*x2;
      F2(x2 + 1);
  }
}
```

MS(F1, F2): (x1 = x2 && g1 <= g2 && x1 >= 0) ==> g1’ <= g2’

- What is a mutual summary MS(F1, F2)?
  - A specification over **two-procedures’** *input/output* vocabulary
    - parameters, globals (g), returns and next state of globals (g’)

Mutual summaries

```
void F1(int x1){
  if(x1 < 100){
      g1 := g1 + x1;
      F1(x1 + 1);
  }
}
```

```
void F2(int x2){
  if(x2 < 100){
      g2 := g2 + 2*x2;
      F2(x2 + 1);
  }
}
```

MS(F1, F2): (x1 = x2 && g1 <= g2 && x1 >= 0) ==> g1’ <= g2’

- When does procedure pair (F1,F2) satisfy MS(F1, F2)?
  - For any (pre,post) state pairs (s1,s1’) of F1, and (s2,s2’) of F2, (s1,s1’,s2,s2’) satisfies MS(F1,F2)

Factorial (revisited)

```
f1(n): returns r {
    if (n == 0) {
        return 1;
    } else {
        return n * f1(n - 1);
    }
}
main(n) : r {r := f1(n);}
```

```
f2(n, a) : returns r {
    if (n == 0) {
        return a;
    } else {
        return f2(n - 1, a * n);
main(n) : r {r := f2(n,1);}
```

MS(f1, f2):
(n1 == n2) ==> (r1*a2 == r2)

procedure MS_check_main_main'(n, n'):
(r, r');
ensures (n == n' ==> r == r')

Note: Splitting a MS check

When MS(i,i‘,o,o’) is of the form

MS_pre(i,i‘) ==>MS_post(o,o’)

The following sound check avoids disjunction in specifications (less efficient to infer)

```
procedure MS_Check_p_p’(i,i’) : (o, o’);
requires MS_pre(i,i‘);
ensures MS_post(o,o’);
```

Differential verification

(Modular) verification problem

- Given a program P
  - A list of procedures p1, p2, …
  - Each procedure has assert , requires, ensures
- Verify that each procedure satisfies its specifications/contracts (assuming the contracts of other procedures)

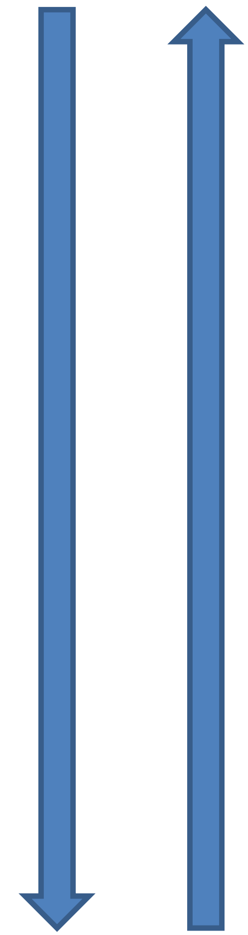
(Modular) differential verification problem

- Given two programs P and P’
  - A list of procedures {p1, p2, …} and {p1’, p2’, ..}
  - Mutual summary specifications MS(p,q’), where (p,q’) \in P X P’
    - Need not be 1-1
- Verify that each MS_Check_p_q’ procedure satisfies its specifications/contracts (assuming the contracts of other procedures)

Sound solutions

- Different product construction (aka proof rules)
- Semantic equivalence (e.g. compiler loop optimizations)
  - [Necula PLDI’00]
- Equivalence with inlining
  - Tries to inline upto recursion when equiv does not hold
  - Useful mostly in the presence of changes in mutually recursive procs
  - [Godlin & Strichman DAC’09]
- Mutual summaries without inference
  - [Hawblitzel, Kawaguchi, Lahiri, Rebelo CADE’13]
- Mutual summaries with invariant inference
  - [Lahiri, McMillan, Sharma, Hawblitzel FSE’13]

More efficient



More precise

Strong semantic equivalence

- Construct the EQ procedures

```
procedure EQ_p_p'(i, i'): (o,o') {
    call o :=  p(i);         //modifies g
    call o' := p'(i');      //modifies g'
}
```

- Perform a bottom up analysis
  - Perform equivalence of p and p' after proving equivalence of callees
  - Make equivalent procedures deterministic uninterpreted functions
- Recursion
  - Sound to *assume* recursive calls to p and p' are equivalent when proving equivalence of p and p'
- Problem
  - Limited applicability
  - Mismatched parameters
  - More complex differential invariants

Mutual summaries with invariant inference

  - [S. Lahiri, K. McMillan, R. Sharma, C. Hawblitzel FSE'13]
- Two steps
  - Convert the differential verification problem to a single program verification problem
  - Leverage *any* program verification technique to infer invariants on MS_check_f_f' procedures

- Why can't we infer invariants on EQ_f_f' procedure described earlier?
  - Because we did not have any callers for these special procedures

Product Program

```
proc f1(x1): r1
modifies g1
{
   s1;
L1:
   w1 := call h1(e1);
   t1
}
```

```
proc f2(x2): r2
modifies g2
{
   s2;
L2:
   w2 := call h2(e2);
   t2
}
```

Instrument calls

Instrument calls

Replay, constrain, restore

```
proc    f1_f2(x1,x2) returns (r1,r2)
modifies g1, g2
{
  // initialize call witness variables
  b_l1, b_l2, ... := false, false, ...;

  [[s1 ;]]
L1:
  i_l1, gi_l1 := e1, g1 ; //store inputs
  call w1 := h1(e1);
  b_l1 := true; //set call witness
  o_l1, go_l1 := w1, g1; //store outputs

  [[t1 ;]]

  [[s2 ;]]
L2:
  i_l2, gi_l2 := e2, g2 ; //store inputs
  call w2 := h2(e2);
  b_l2 := true; //set call witness
  o_l2, go_l2 := w2, g2; //store outputs

  [[t2 ;]]

  //one block for each pair of call sites
  //for a pair of mapped procedures
  ....
  if (b_l1 && b_l2) { //for (L1,L2) pair
    //store the globals
    st_g1, st_g2 := g1, g2;

    g1, g2 := gi_l1, gi_l2 ;
    call k1, k2 :=    h1_h2(i_l1, i_l2 );
    assume (k1 == o_l1 && g1 == go_l1);
    assume (k2 == o_l2 && g2 == go_l2);

    //restore globals
    g1, g2 := st_g1, st_g2;
  }
  ...

  return;
}
```

f1

f2

Reduce differential verification ➔ single program verification

```
proc f1(x1): r1
modifies g1
{
   s1;
L1:
   w1 := call h1(e1);
   t1
}
```

```
proc f2(x2): r2
modifies g2
{
   s2;
L2:
   w2 := call h2(e2);
   t2
}
```

Novel product construction

```
proc     f1_f2(x1,x2) returns (r1,r2)
modifies g1, g2
{
   // initialize  call  witness  variables
   b_l1, b_l2, ... := false, false, ...;

   [[s1 ;]]
L1:
   i_l1, gi_l1  := e1, g1 ; //store inputs
   call w1 := h1(e1);
   b_l1 := true; //set call witness
   o_l1, go_l1 := w1, g1; //store outputs

   [[t1 ;]]

   [[s2 ;]]
L2:
   i_l2, gi_l2  := e2, g2 ; //store inputs
   call w2 := h2(e2);
   b_l2 := true; //set call witness
   o_l2, go_l2 := w2, g2; //store outputs

   [[t2 ;]]

   //one block for each pair of call sites
   //for a pair of mapped procedures
   ....
   if (b_l1 && b_l2) { //for (L1,L2) pair
      //store the globals
      st_g1, st_g2 := g1, g2;

      g1, g2 := gi_l1, gi_l2;
      call k1, k2 :=    h1_h2(i_l1, i_l2 );
      assume (k1 == o_l1 && g1 == go_l1);
      assume (k2 == o_l2 && g2 == go_l2);

      //restore globals
      g1, g2 := st_g1, st_g2;
   }
   ...

   return;
}
```

Off-the-shelf program verifier + invariant inference

Properties

- A little formalism first

- For a procedure p,
  - TR(p) = {(i,o) |exists an execution from input state i to output state o} //transition relation
  - For a postcondition S of p
    - ||S|| = {(i,o) | all input/output state pairs that make S true}
  - p *satisfies* S if TR(p) $\subseteq$ ||S||
- Applies even to MS_check_p_p’ procedures
  - MS_check_p_p’ *satisfies* MS(p,p’) if TR(MS_check_p_p’) $\subseteq$ || MS(p,p’) ||

Property

Theorems:

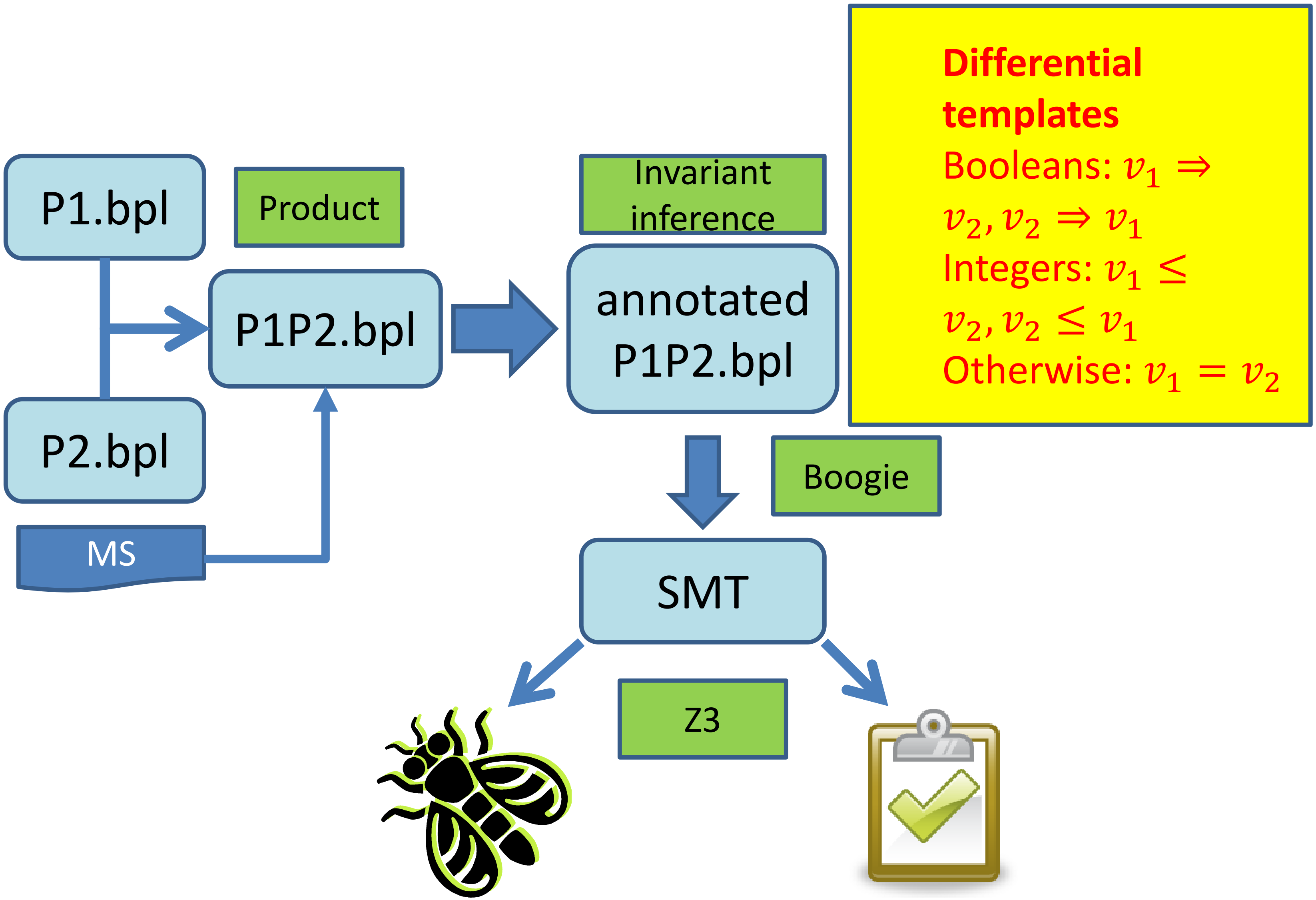
- If each MS_check_p_p' *modularly satisfies* MS(p,p'), then each MS_check_p_p' *satisfies* MS(p,p')

- It allows us to infer invariants treating MS_check_p_p' as a single program

Automatic differential invariant inference

- Exploit the structural similarity between programs
  - Provide simple differential predicates (difficult to infer by program verification tools such as iZ3)
  - Predicates x <> x’, where x in p and x’ in p’, and <> ∈ {==, <=, >=, ==>, …}
- Predicate Abstraction [Graf&Saidi ‘95]
  - Infer Boolean combination of predicates
  - Can efficiently infer subsets of predicates that hold (Houdini)

Implementation Workflow



SymDiff Applications

- Differential memory safety for buffer bounds bugfixes
- Proving approximate transformations safe
- Cross-version compiler validation of CLR
  - [Hawblitzel, Lahiri et al. FSE’13, Lahiri et al. CAV’15]
- Translation validation of compiler loop optimizations
- Ironclad informational flow checking
  - [Hawblitzel et al. OSDI ‘14]

Verifying Bug Fixes

- Does a fix inadvertently introduce new bugs?

- Verisec suite:

“snippets of open source programs which contain buffer overflow vulnerabilities, as well as corresponding patched versions.”

- Relative buffer overflow checking

- Examples include apache, madwifi, sendmail, …

Stringcopy (revisited)

```
void strcopy_buggy
(char* dst, char*src, int size)
{
  int i=0;
  for(;*src && i<size-1; i++)
        *dst++ = *src++;
  *dst = 0;
}
```

```
void strcopy_correct
(char* dst, char*src, int size)
{
  int i=0;
  for(;i<size-1 && *src; i++)
        *dst++ = *src++;
  *dtmp = 0;
}
```

Can prove relative memory-safety automatically

- No preconditions required
- Assertion does not need to know the buffer length!

Relative invariants:
src.1=src.2, dst.1=dst.2, size.1=size.2, i.1=i.2, ok.1 ==> ok.2

Example

```
int main_buggy()
{
  …
  fb := 0;
  while(c1=read()!=EOF)
  {
    fbuf[fb] = c1;
    fb++;

  }
  …
}
```

Buffer Overflow

```
int main_patched()
{
  …
  fb := 0;
  while(c1=read()!=EOF)
  {
    fbuf[fb] = c1;
    fb++;
    if(fb >= MAX)
      fb = 0;
  }
}
```

Invariant: fb.2<=fb.1

Safety of approximate transformations

- Programmer may sacrifice some precision to optimize performance
  - Multimedia applications, search results
  - Programmers can control which part of the program/data is stored in approximate but faster hardware (more prone to faults)

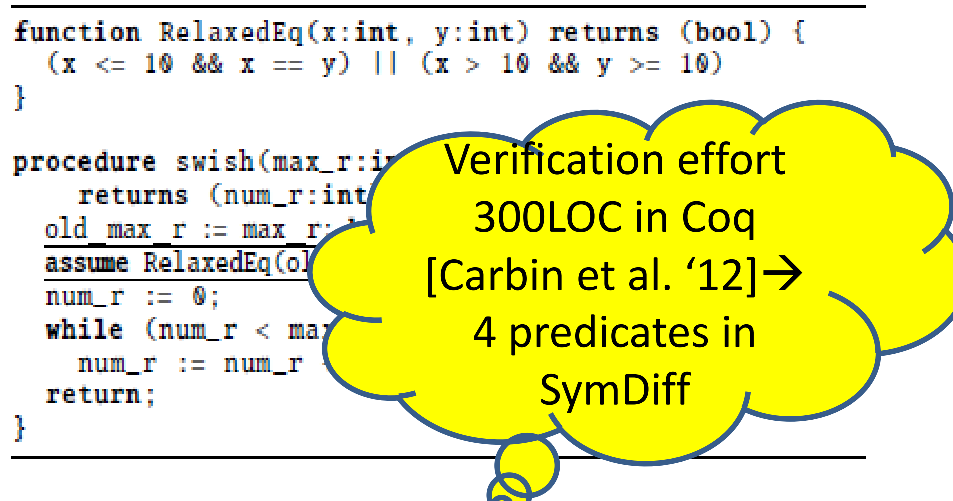


Fig. 2: Swish++ example with dynamic knobs approximation.

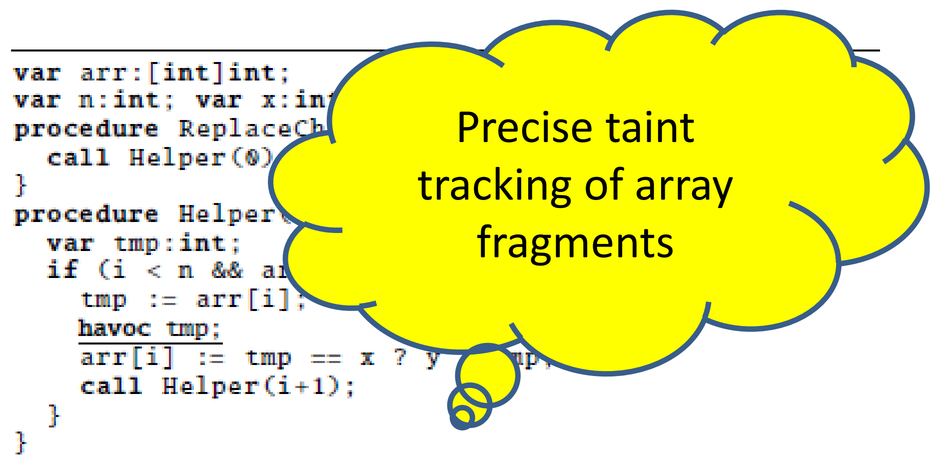


Fig. 1: Replacing a character in a string.

Lahiri, Haran, He, Rakamaric MSRTR 2015

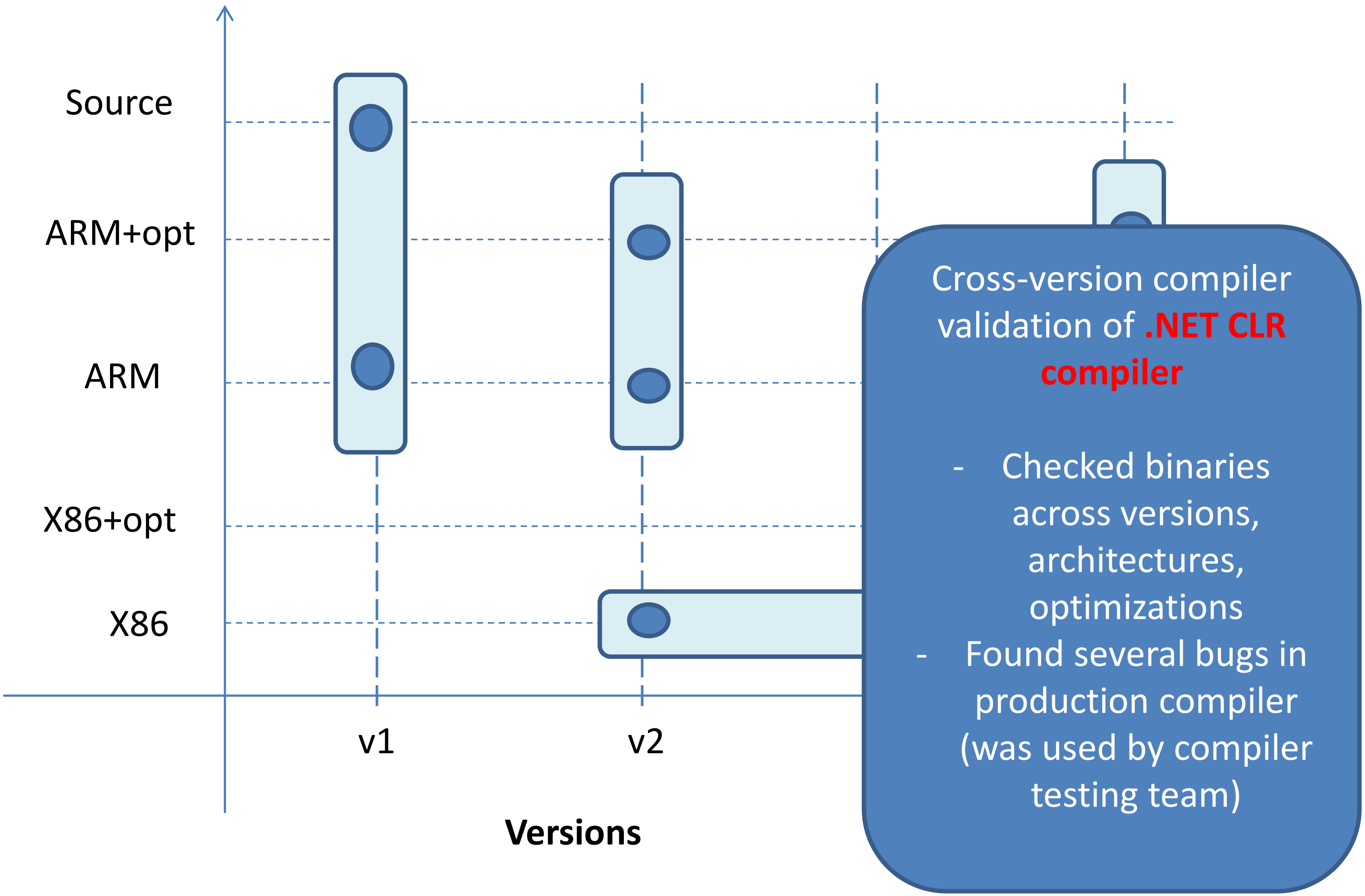
Outline

- ✓ Motivation
- ✓ SymDiff: A differential program verifier
  - Program verification background
  - Differential specifications
  - Differential program verification
- SymDiff: Applications
- Other applications of differential reasoning for existing verifiers
  - Verification modulo versions, Interleaved bugs
- Other works in differential cross-version program analysis
- Works in differential analysis of independent implementations

SymDiff Applications

- ✓ Differential memory safety for buffer bounds bugfixes
- ✓ Proving approximate transformations safe
- Cross-version compiler validation of CLR
  - [Hawblitzel, Lahiri et al. FSE’13, Lahiri et al. CAV’15]
- Translation validation of compiler loop optimizations
- Ironclad informational flow checking
  - [Hawblitzel et al. OSDI ‘14]

Compiler validation



Compatibility: x86 vs. x86 example

C:\x\qj3\SemanticDiff\bin\test\trace\fn — AnalyzeTraces

```
1:   ; Assembly listing for method System.
Windows. FrameworkElement:
set_SubtreeHasLoadedChangeHandler(bool)
2:   ; Emitting BLENDED_CODE for Pentium 4
3:   ; optimized code
4:   ; ESP based frame
5:   ; partially interruptible
6:   ; Final local variable assignments
7:   ;
8:   ; V00 this     [V00,T00] ( 3,  3 )   ref
-> ECX    this
9:   ; V01 arg1     [V01,T01] ( 3,  3 )  bool
-> EAX
10:
11:   G_M63730_IG01:
12:         mov   EAX, EDX
13:
14:   G_M63730_IG02:
15:         and   EAX, 255
      [eax = 181]
16:         push  EAX
      [stored_value = 181]
17:         mov   EDX, 0x100000
18:         call  System. Windows.
FrameworkElement: WriteInternalFlag2
(int,bool)
```

```
Windows. FrameworkElement:
set_SubtreeHasLoadedChangeHandler(bool)
2:   ; Emitting BLENDED_CODE for Pentium 4
3:   ; optimized code
4:   ; esp based frame
5:   ; partially interruptible
6:   ; Final local variable assignments
7:   ;
8:   ; V00 this     [V00,T00] ( 3,  3 )   ref
-> ecx    this
9:   ; V01 arg1     [V01,T01] ( 3,  3 )  bool
-> esi
10:
11:   G_M57940_IG01:
12:         push  ESI
13:         mov   ESI, EDX
14:
15:   G_M57940_IG02:
16:         and   ESI, 254
      [esi = 111]
17:         push  ESI
      [stored_value = 111]
18:         mov   EDX, 0x100000
19:         call  System. Windows.
FrameworkElement: WriteInternalFlag2
(int,bool)
```

possible cause: argument 3 (Mem[esp+0]) differs:

Large x86 vs. ARM example

Translation validation of compiler loop optimizations

- Looked at translation validation of parameterized programs [Kundu, Tatlock, Lerner '09]
- Manual mutual summaries (to test the extent of mutual summaries)

- Optimizations that can be proved
  - Copy propagation, constant propagati elimination, partial redundancy elin hoisting, conditional speculation, spe loop unswitching, loop unrolling, loop p

Reasonable since manual changes are seldom as complex

- Optimizations that can't be proved
  - Loop alignment, loop interchange, loop reversal, loop skewing, loop fusion, loop distribution
  - Reason: the order of updates to array indices differ
  - Previous works need a PERMUTE rule specific to reorder loop iterations [Zuck et al. '05]

Outline

- ✓ Motivation
- ✓ SymDiff: A differential program verifier
  - ✓ Program verification background
  - ✓ Differential specifications
  - ✓ Differential program verification
- ✓ SymDiff: Applications
- Other applications of differential reasoning for existing verifiers
  - Verification modulo versions, Interleaved bugs
- Other works in differential cross-version program analysis
- Works in differential analysis of independent implementations

Diff verif for existing verifiers

- Program verifiers suffer from false alarm due to under constrained environments (stubs, inputs)

- Verification Modulo Versions (VMV)
  - [Logozzo, Lahiri, Fahndrich, Blackshear PLDI'14]
  - Necessary and sufficient conditions to give relative guarantees, or point regressions (based on abstract interpretation)
  - Integrated with production static analyzer Clousot, verifying 80% of alarms for relative correctness

- Interleaved bugs for concurrent programs
  - [Joshi, Lahiri, Lal POPL'12]
  - Using coarse interleavings as a specification to tolerate environment imprecision
  - Applied on concurrent device drivers in Windows

Related works in cross-version program analysis

- Regression verification [Godlin & Strichman DAC’09,..]
- Differential symbolic execution [Person et al. FSE’08,..], DiSE [Person et al. PLDI’12]
- Abstract differencing using abstract interpreters [Partush et al. ’13]
- UC-KLEE [Ramos & Engler CAV’11]
- Change contracts [Yi et al. ISSTA’13]

Other examples of differential analysis of independent implementations

- Compiler testing
  - Translation validation [Pnueli et al.’98, Necula ’00,…]
  - Differential compiler testing [Regehr et al. PLDI’11, ..]
- Security testing
  - Java security APIs vulnerabilities [Srivastava et al. PLDI’11]
  - SSL/TLS certificate validation [Brubaker et al. S&P’14]
  - String validation in web applications[Alkhalaf et al. ISSTA’14]

Outline

- ✓ Motivation
- ✓ SymDiff: A differential program verifier
  - ✓ Program verification background
  - ✓ Differential specifications
  - ✓ Differential program verification
- ✓ SymDiff: Applications
- ✓ Other applications of differential reasoning for existing verifiers
  - ✓ Verification modulo versions, Interleaved bugs
- ✓ Other works in differential cross-version program analysis
- ✓ Works in differential analysis of independent implementations

Summary

A framework to

- Leverage and extend program verification for *differential verification*

Source code

http://symdiff.codeplex.com/

Papers etc.

http://research.microsoft.com/symdiff

Research questions

- Relative termination
- Semantic change impact analysis
- Adding probabilistic reasoning
- Other generic relative specifications
- Diff verification of concurrent programs