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.

Does the **refactoring** change any observable behavior?

How does the **feature addition** impact existing features?

Does my **bug-fix** introduce a regression?
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
  - f() { Print(foo); g(); }
  - g() { ... Print(foo); } → g() { ... Print(foo); Print(bar); }
- New features
- Version Control
- Library API changes
- Compilers
- 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
- 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
• 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
  - \( x[i] := y \rightarrow x := upd(x, i, y) \)
  - \( y := x[i] \rightarrow y := sel(x,i) \)

- \texttt{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;
```

```
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;
  ```
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(u: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

\{ 
  \text{start: } x := 1; \text{ goto } l_1; 
\}

B

\{ 
  l_1: x := x + 1; \text{ goto } l_2, l_3; 
\}

C

\{ 
  l_2: \text{ assume } x == 0; 
  x := x + 2; 
  \text{ goto } l_4; 
\}

D

\{ 
  l_3: \text{ assume } x \neq 0; 
  x := x + 3; 
  \text{ goto } l_4; 
\}

E

\{ 
  l_4: \text{ 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} \iff (x_0 == 1 \Rightarrow B_{ok})

B_{ok} \iff (x_0 == 1 \Rightarrow C_{ok} \land D_{ok})

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

D_{ok} \iff (x_1 \neq 0 \Rightarrow (x_2 == x_1 + 3 \Rightarrow (x_4 == x_3 \Rightarrow E_{ok})))

E_{ok} \iff (x_4 == 5 \land true)

\Rightarrow A_{ok}
VC Generation

Formula over Arithmetic, Equality, and Boolean connectives

Can be solved by a SMT solver

\[
A_{ok} \iff (x_0 == 1 \implies B_{ok})
\]
\[
B_{ok} \iff (x_0 == 1 \implies C_{ok} \land D_{ok})
\]
\[
C_{ok} \iff (x_1 == 0 \implies
(x_2 == x_1 + 2 \implies
(x_4 == x_2 \implies E_{ok}))
)
\]
\[
D_{ok} \iff (x_1 \neq 0 \implies
(x_2 == x_1 + 3 \implies
(x_4 == x_3 \implies E_{ok}))
)
\]
\[
E_{ok} \iff (x_4 == 5 \land \text{true})
\]
\[
\implies A_{ok}
\]
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** \( \text{EQ\_p\_p'} \)

```plaintext
procedure EQ\_p\_p'(i, i'): (o,o') {
    call o := p(i);       //modifies g
    call o' := p'(i');   //modifies g'
}
```

- Write a postcondition
  - ensures \((i == i' && \text{old}(g) == \text{old}(g') \Longrightarrow o == o')\)
  - ensures \((i == i' && \text{old}(g) == \text{old}(g') \Longrightarrow g == g')\)

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

\[ f_1(n) : \text{returns } r \{ \]
\[
\quad \text{if } (n == 0) \{ \]
\[
\quad \quad \text{return } 1; \]
\[
\quad \}\text{ else } \{ \]
\[
\quad \quad \text{return } n * f_1(n - 1); \]
\[
\}\]
\[
\}
\[
\text{main}(n) : r \{ r := f_1(n); \}
\]

\[ f_2(n, a) : \text{returns } r \{ \]
\[
\quad \text{if } (n == 0) \{ \]
\[
\quad \quad \text{return } a; \]
\[
\quad \}\text{ else } \{ \]
\[
\quad \quad \text{return } f_2(n - 1, a * n); \]
\[
\}\]
\[
\}
\[
\text{main}(n) : r \{ r := f_2(n, 1); \}
\]

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 $\text{EQ}_p_p’$ procedure
  – Replace $\text{assert } A \rightarrow ok := ok \&\& A$;
  – Write a postcondition
    $\text{ensures } (i == i’ \&\& \text{old}(g) == \text{old}(g’) \rightarrow (ok \rightarrow ok’))$

• Note: asymmetric check
Relative Correctness (fails)

```c
void strcopy_buggy (char* dst, char*src, int size)
{
    int i = 0;
    for(;*src; 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++;
    *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++;
    *dst = 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

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’)

```c
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’
Mutual summaries

When does procedure pair \((F_1,F_2)\) satisfy \(\text{MS}(F_1,F_2)\)?

- For any \((\text{pre},\text{post})\) state pairs \((s_1,s_1')\) of \(F_1\), and \((s_2,s_2')\) of \(F_2\), \((s_1,s_1',s_2,s_2')\) satisfies \(\text{MS}(F_1,F_2)\)

\[
\begin{align*}
\text{void F1(int x1)} & \{ \\
\text{if(x1 < 100)} & \{ \\
& \text{g1 := g1 + x1;} \\
& \text{F1(x1 + 1);} \\
& \} \\
& \} \\
\text{void F2(int x2)} & \{ \\
\text{if(x2 < 100)} & \{ \\
& \text{g2 := g2 + 2*x2;} \\
& \text{F2(x2 + 1);} \\
& \} \\
& \} \\
\end{align*}
\]

\(\text{MS}(F_1, F_2): \ (x_1 = x_2 \land g_1 \leq g_2 \land x_1 \geq 0) \implies g_1' \leq g_2'\)
Factorial (revisited)

\[ f_1(n) : \text{returns } r \{
    \text{if } (n == 0) \{
        \text{return } 1;
    \}
    \text{else } \{
        \text{return } n \ast f_1(n - 1);
    \}
\} \]

\[ main(n) : r \{ \text{r := } f_1(n); \} \]

\[ f_2(n, a) : \text{returns } r \{
    \text{if } (n == 0) \{
        \text{return } a;
    \}
    \text{else } \{
        \text{return } f_2(n - 1, a \ast n);
    \}
\} \]

\[ main(n) : r \{ \text{r := } f_2(n,1); \} \]

\[ \text{procedure } \text{MS\_check\_main\_main}'(n, n'):\]
\[ (r, r'); \]
\[ \text{ensures } (n == n' ==> r == r') \]

\[ \text{MS}(f_1, f_2):\]
\[ (n_1 == n_2) ==> (r_1 \ast a_2 == r_2) \]
Note: Splitting a MS check

When $MS(i,i',o,o')$ is of the form

$$MS_{\text{pre}}(i,i') \Rightarrow MS_{\text{post}}(o,o')$$

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

```plaintext
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 \{p_1, p_2, \ldots\} and \{p_1’, p_2’, \ldots\}
  – Mutual summary specifications \text{MS}(p,q’), where (p,q’) \in P \times P’
    • Need not be 1-1

• Verify that each \text{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 up to 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]
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**

```plaintext
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

Instrument calls

Instrument calls

Instrument calls
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
}

Reduce differential verification ➔ single program verification

Novel product construction

Off-the-shelf program verifier + invariant inference
Properties

– A little formalism first

• For a procedure p,
  – $TR(p) = \{(i,o) \mid \text{exists an execution from input state i to output state o}\}$ //transition relation

  – For a postcondition S of p
    • $||S|| = \{(i,o) \mid \text{all input/output state pairs that make S true}\}$

  – 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 $<> \in \{==, <=, >=, ==>, \ldots\}$

• Predicate Abstraction [Graf&Saidi ‘95]
  – Infer Boolean combination of predicates
  – Can efficiently infer subsets of predicates that hold (Houdini)
Implementation Workflow

P1.bpl  Product  P1P2.bpl  annotated P1P2.bpl

P2.bpl

MS

Invariants inference

Differential templates
- Booleans: $v_1 \Rightarrow v_2, v_2 \Rightarrow v_1$
- Integers: $v_1 \leq v_2, v_2 \leq v_1$
- Otherwise: $v_1 = v_2$

Boogie

SMT

Z3

MS
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)

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

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

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)

```
function RelaxedEq(x:int, y:int) returns (bool) {
  (x <= 10 && x == y) || (x > 10 && y >= 10)
}

procedure swish(max_r:int)
returns (num_r:int)
old_max_r := max_r;
assume RelaxedEq(old_max_r);
num_r := 0;
while (num_r < max_r)
  num_r := num_r + 1;
return;
}
```

Verification effort
300LOC in Coq
[Carbin et al. ‘12] → 4 predicates in SymDiff

```
var arr:[int]int;
var n:int; var x:int;
procedure ReplaceChar();
call Helper(0);
}
procedure Helper(i:int); var tmp:int;
if (i < n && arr[i] == x)
  tmp := arr[i];
  havoc tmp;
  arr[i] := tmp == x ? y : arr[i];
call Helper(i+1);
```

Precise taint tracking of array fragments

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]
Cross-version compiler validation of .NET CLR compiler

- Checked binaries across versions, architectures, optimizations
- Found several bugs in production compiler (was used by compiler testing team)
Compatibility: x86 vs. x86 example

```
1: ; Assembly listing for method System.
2: Windows. FrameworkElement:
3: set_SubtreeHasLoadedChangeHandler(bool)
4: ; Emitting BLENDED_CODE for Pentium 4
5: ; optimized code
6: ; ESP based frame
7: ; partially interruptible
8: ; Final local variable assignments
9: ;
10: G_M63730_IG01:
11: _ mov EAX, EDX
12: G_M63730_IG02:
13: _ and EAX, 255
14: [eax = 101]
15: _ push EAX
16: [stored_value = 181]
17: _ mov EDX, 0x100000
FrameworkElement: WriteInternalFlag2
(int,bool)
```

```
11: G_M57940_IG01:
12: _ push ESI
13: _ mov ESI, EDX
14: _
15: G_M57940_IG02:
16: _ and ESI, 254
17: [esi = 111]
18: _ push ESI
19: [stored_value = 111]
```

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 propagation, common subexpression elimination, partial redundancy elimination, loop invariant code hoisting, conditional speculation, software pipelining, loop unswitching, loop unrolling, loop peeling

• 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]

Reasonable since manual changes are seldom as complex
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