Interprocedural Analysis

17-654/17-754

Analysis of Software Artifacts

Jonathan Aldrich

Interprocedural Analysis Strategies

- Make default assumptions
- Assume and check annotations
- Build Interprocedural CFG
- Compute Summaries
  - All contexts one by one
  - Each context on demand
  - All context at once

Making Default Assumptions

- Assumptions
  - Starting dataflow value for all parameters
  - Dataflow value for result
  - Starting and ending information for globals if you're tracking them (*most analyses don't*)
- Verification
  - Initial info: starting value for parameters
  - Verify result $\sqsubseteq$ assumption$_{result}$
    - Ending value for result obeys assumption
  - Verify arg $\sqsubseteq$ assumption$_{arg}$
    - Actual arguments obey assumptions of formal parameter
  - Similar extension to globals if you're tracking them

Making Default Assumptions

- Example: Zero Analysis
  - Default: $\top$ (MZ) for arguments and results
    - Benefit: actual arguments and actual result always obey assumption
    - Cost: very conservative for arguments
      - Will report false positive errors
  - Globals: easiest solution is not to track them
    - Could also track but assume $\top$ at boundaries

Example: Default Assumptions

```
int divByX(int x) {
   [result := 10/x]1;
}

void caller() {
   [x := 5]1;
   [y := divByX(x)]2;
}
```

- Analyze divByX

p	x	result
0	MZ	MZ
1	MZ	NZ

  - Warning: div by zero at 1
  - Verify σ[result] ⊑ MZ
- Analyze caller

p	x	y
0	MZ	MZ
1	NZ	MZ
2	NZ	MZ

  - Verify σ[x] ⊑ MZ
- Note that div by zero can't happen!

Optimistic Assumption: NZ

```
int divByX(int x) {
   [result := 10/x]1;
}

void caller() {
   [x := 5]1;
   [y := divByX(x)]2;
}
```

- Analyze divByX

p	x	result
0	NZ	MZ
1	NZ	NZ

  - No warning
  - Verify σ[result] ⊑ NZ
- Analyze caller

p	x	y
0	MZ	MZ
1	NZ	MZ
2	NZ	NZ

  - Verify σ[x] ⊑ NZ

Optimistic Assumption: NZ

```
int double(int x) {
   [result := 2*x]1;
}

void caller() {
   [x := 0]1;
   [y := double(x)]2;
}
```

- Analyze double

p	x	result
0	NZ	MZ
1	NZ	NZ

  - No warning
  - Verify σ[result] ⊑ NZ
- Analyze caller

p	x	y
0	MZ	MZ
1	Z	MZ
2	Z	NZ

  - Verify σ[x] ⊑ NZ fails!
  - False positive—this code is OK

Assume and Check Annotations

- Annotations
  - Starting dataflow value for all parameters
  - Dataflow value for result
- Verification
  - Initial info: starting value for parameters
  - Verify result ⊑ annotation$_{result}$
    - Ending value for result obeys annotation
  - Verify arg ⊑ annotation$_{arg}$
    - Actual arguments obey annotations on formal parameter

Assumption Example

```
@NZ int divByX(@NZ int x) {
   [result := 10/x]1;
}

void caller() {
   [x := 5]1;
   [y := divByX(x)]2;
}
```

- Analyze divByX

p	x	result
0	NZ	MZ
1	NZ	NZ

- Verify σ[result] ⊑ NZ
- Analyze caller

p	x	y
0	MZ	MZ
1	NZ	MZ
2	NZ	NZ

- Verify σ[x] ⊑ NZ

Assumption Example

```
@MZ int double(@MZ int x) {
   [result := 2*x]1;
}

void caller() {
   [x := 5]1;
   [y := double(x)]2;
   [z := 10/y]3;
}
```

- Analyze divByX

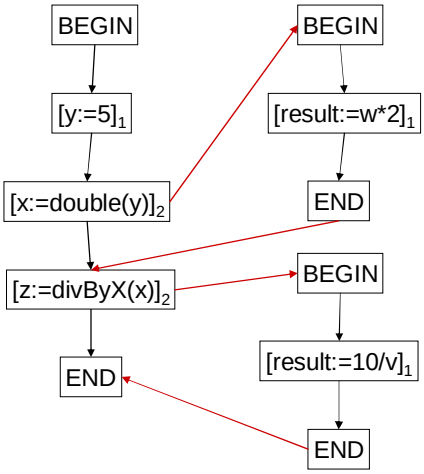
p	x	result
0	MZ	MZ
1	MZ	MZ

- Verify σ[result] ⊑ MZ
- Analyze caller

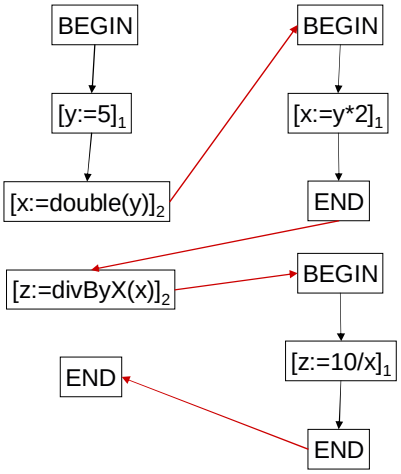
p	x	y	z
0	MZ	MZ	MZ
1	Z	MZ	MZ
2	Z	MZ	MZ
3	Z	MZ	MZ

- Verify σ[x] ⊑ MZ
- Warning: possible div by zero
- False positive!

Interprocedural CFG Intuition



Interprocedural CFG Intuition



Interprocedural CFG Example

```
int double(int x) {
   [y := 2*x]6;
}

void caller() {
   [x := 5]1;
   [y := double(x)]2;
   [z := 10/y]3;
   [x := 0]4;
   [y := double(x)]5;
}
```

p	x	y	z
0	MZ	MZ	MZ
1	NZ	MZ	MZ
6	NZ	NZ	MZ
2	NZ	NZ	MZ
3	NZ	NZ	NZ

- No div by zero

4	Z	NZ	NZ
6	MZ	MZ	MZ
5	MZ	MZ	MZ

- Must revisit node 2 because result of double changed

2	MZ	MZ	MZ
3	MZ	MZ	NZ

- Divide by zero warning
- False positive!

Context Sensitive Summaries

- Intuition
  - Interprocedural CFG loses too much precision when a function is called with different argument dataflow lattice elements
  - Simple annotations have same issue
    - (but same Summary technique works there)
- Summaries
  - Maps from input dataflow information to output dataflow information
  - *Context sensitive*: different results for different calls
  - When function is called, apply the map!

Generating Context Sensitive Summaries

- Brute force
  - Analyze the function once for each possible input lattice element
  - Problem: way too many lattice elements—would take too long
- On demand
  - Analyze the function once for each actual input lattice element it is called with
  - Much better—but can still be impractical for large programs with precise lattices
- Abstract summaries
  - Symbolically represent function's effect on input lattice element
  - Example: PREfix's technique
  - The state of the art in interprocedural analysis

On Demand Summaries

```
/* Summary
 * Case x:NZ -> result:NZ

*/
int double(int x) {
   [result := 2*x]1;
}

void caller() {
   [x := 5]1;
   [y := double(x)]2;
   [z := 10/y]3;
   [x := 0]4;
   [y := double(x)]5;
}
```

p	x	y	z
0	MZ	MZ	MZ
1	NZ	MZ	MZ
2	NZ	NZ	MZ

Compute summary of double for x:NZ

p	x	result
0	NZ	MZ
1	NZ	NZ

On Demand Summaries

```
/* Summary
 * Case x:NZ -> result:NZ
 * Case x:Z -> result:Z
*/
int double(int x) {
    [result := 2*x]1;
}

void caller() {
    [x := 5]1;
    [y := double(x)]2;
    [z := 10/y]3;
    [x := 0]4;
    [y := double(x)]5;
}
```

p	x	y	z
0	MZ	MZ	MZ
1	NZ	MZ	MZ
2	NZ	NZ	MZ
3	NZ	NZ	NZ
4	Z	NZ	NZ
5	Z	Z	NZ

Compute summary of double for x:Z

p	x	result
0	Z	MZ
1	Z	Z

Context Sensitive Annotations

```
@Case("x:NZ -> result:NZ")
@Case("x:Z -> result:Z")
int double(int x) {
    [result := 2*x]1;
}

void caller() {
    [x := 5]1;
    [y := double(x)]2;
    [z := 10/y]3;
    [x := 0]4;
    [y := double(x)]5;
}
```

Verify annotation @Case("x:Z -> result:Z")

p	x	result
0	Z	MZ
1	Z	Z

Verify annotation @Case("x:NZ -> result:NZ")

p	x	result
0	NZ	MZ
1	NZ	NZ

Verify client

p	x	y	z
0	MZ	MZ	MZ
1	NZ	MZ	MZ
2	NZ	NZ	MZ *// case x:NZ*
3	NZ	NZ	NZ
4	Z	NZ	NZ
5	Z	Z	NZ *// case x:Z*

Abstract Summaries

```
/* Summary
 * Case x:α -> result:α
 */
int double(int x) {
   [result := 2*x]1;
}

void caller() {
   [x := 5]1;
   [y := double(x)]2;
   [z := 10/y]3;
   [x := 0]4;
   [y := double(x)]5;
}
```

Compute summary of double for x:α

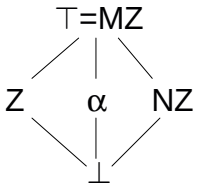
p	x	result
0	α	MZ
1	α	α

Analyze client

p	x	y	z
0	MZ	MZ	MZ
1	NZ	MZ	MZ
2	NZ	NZ	MZ // α=*NZ*
3	NZ	NZ	NZ
4	Z	NZ	NZ
5	Z	Z	NZ // α=*Z*

Abstract Summaries for Zero Analysis

- New ZA lattice has α
- Flow functions
  - $f_{ZA}(\sigma, [x]_k) = [t_k \mapsto \sigma(x)]\ \sigma$
  - $f_{ZA}(\sigma, [n]_k)$ = if n==0
    then $[t_k \mapsto Z]\sigma$
    else $[t_k \mapsto NZ]\sigma$
  - $f_{ZA}(\sigma, [x := \ldots]_k) = [x \mapsto \sigma(t_n)]\ \sigma$
  - $f_{ZA}(\sigma, [[\ldots]_n\ op\ [\ldots]_m]_k)$ =
    if op=* and $\sigma[t_m]$=NZ
    then $[t_k \mapsto \sigma(t_n)]\sigma$ *// this case used in example*
    if …
    else $[t_k \mapsto MZ]\ \sigma$
  - $f_{ZA}(\sigma$, */* any other */*$) = \sigma$
- Many other ways to generate summaries



Comparison

- Assumptions
  - Simple, efficient
  - Imprecise
- Annotations
  - Require effort
  - More precise than assumptions
  - More efficient than IP analysis
  - Can used "summary annotations" to get context sensitivity
- Both work on partial programs
- Interprocedural CFG
  - Simple for programmer
  - As precise as simple annotations
  - Still imprecise, can be very costly
    - $O(n^3)$ in size of program
- Summaries
  - Excellent precision
  - Costly if not abstract
- Both require whole program

Metal: User-Defined Property Checking

Reading: ***Checking System Rules Using System-Specific, Programmer-Written Compiler Extensions***

17-654/17-754
Analysis of Software Artifacts
Jonathan Aldrich

Find the Bug!

```
/* From Linux 2.3.99 drivers/block/raid5.c */
static struct buffer_head *
get_free_buffer(struct stripe_head *sh,
                int b_size) {
        struct buffer_head *bh;
        unsigned long flags;

        save_flags(flags);
        cli();
        if ((bh = sh->buffer_pool) == NULL)
                return NULL;
        sh->buffer_pool = bh->b_next;
        bh->b_size = b_size;
        restore_flags(flags);
        return bh;
}
```

disable interrupts

ERROR: returning with interrupts disabled

re-enable interrupts

Metal

- Strengths?
- Weaknesses?

Metal: System-Specific Rules

- Problem similarities
  - Paths complex, hard to catch with inspections or testing
- Solution similarities
  - Like Fluid, require model of rule
  - Like PREfix, check rules automatically
  - Like PREfix, focus on finding bugs, not assurance
  - Catch violations with flow analysis
- Differences
  - Rules are specific to a system; can't build into tool
  - Analysis is intra-procedural
    - otherwise, too many paths

Example Rules

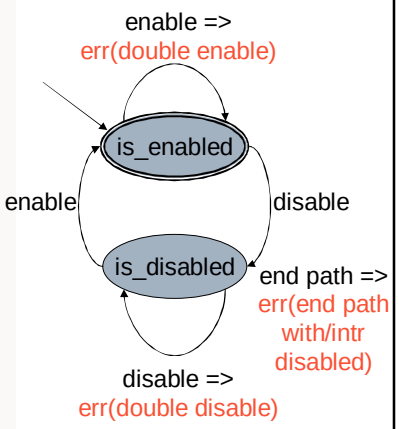
Rule template	Examples
"Never/always do X"	"Do not use floating point in the kernel." (§ 4.3) "Do not allocate large variables on the 6K byte kernel stack." (§ 4.3) "Do not send more than two messages per virtual network lane." "Allocate as much storage as an object needs." (§ 5.2)
"Do X rather than Y"	"Use memory mapped I/O rather than copying." "Avoid globally disabling interrupts."
"Always do X before/after Y"	"Check user pointers before using them in the kernel." (§ 5.1) "Handle operations that can fail (e.g., memory, disk block, virtual interrupt allocation)." (§ 5.2) "Re-enable interrupts after disabling them." (§ 7) "Release locks after acquiring them." (§ 7) "Check user permissions before modifying kernel data structures."
"Never do X before/after Y"	"Do not acquire lock A before B." "Do not use memory that has been freed." (§ 5.2) "Do not (deallocate an object, acquire/release a lock) twice." (§ 5.2 § 7) "Do not increment a module's reference count after calling a function that can sleep." (§ 6.3)
"In situation X, do (not do) Y"	"Protect all variable mutations with write locks." "If a system call fails, reverse all side-effect operations (deallocate memory, disk blocks, pages, unincrement reference counters)." (§ 5.2 § 6.3) "To avoid deadlock, while interrupts are disabled, do not call functions that can sleep." (§ 6.2)
"In situation X, do Y rather than Z"	"If a variable is not modified, protect it with read locks." "If code does not share data with interrupt handlers, then use spin locks rather than the more expensive interrupt disabling." "To save an instruction when setting a message opcode, xor in the new and old opcode rather than using assignment." (§ 8)

Interrupt Checker

```
{ #include "linux-includes.h" }
sm check_interrupts {
  // Variables
  // used in patterns
  decl { unsigned } flags;

  // Patterns
  // to specify enable/disable functions.
  pat enable = { sti(); }
             | { restore_flags(flags); } ;
  pat disable = { cli(); };

  // States
  // The first state is the initial state.
  is_enabled: disable ==> is_disabled
     | enable ==> { err("double enable"); }
     ;
  is_disabled: enable ==> is_enabled
     | disable ==> { err("double disable"); }
     // Special pattern that matches when the SM
     // hits the end of any path in this state.
     | $end_of_path$ ==>
        { err("exiting w/intr disabled!"); }
     ;
}
```



Applying the Interrupt Checker

```
/* From Linux 2.3.99 drivers/block/raid5.c */
static struct buffer_head *
get_free_buffer(struct stripe_head *sh,
                int b_size) {
        struct buffer_head *bh;
        unsigned long flags;

        save_flags(flags);
        cli();
        if ((bh = sh->buffer_pool) == NULL)
                return NULL;
        sh->buffer_pool = bh->b_next;
        bh->b_size = b_size;
        restore_flags(flags);
        return bh;
}
```

initial state is_enabled

transition to is_disabled

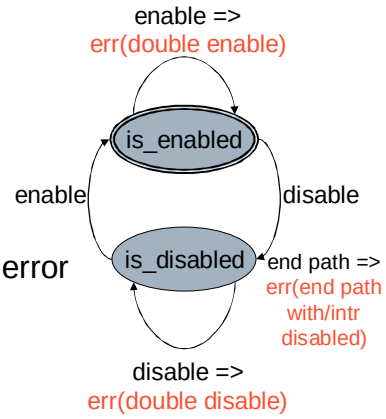
final state is_disabled: ERROR!

transition to is_enabled

final state is_enabled is OK

Checkers

- Not quite a lattice
  - No top/bottom
  - Unnecessary
    - No control flow merges
- Flow functions
  - Pattern matched
    - Go to new state or report error
  - No match
    - Stay in same state



Challenges

- Efficiency and Termination
  - Exponential # paths with if statements
  - Infinite # paths with loops

Challenges

- Efficiency and Termination
  - Exponential # paths with if statements
  - Infinite # paths with loops
- Secret weapon
  - Finite number of states *(see model checking!)*
  - If you come to a statement and you've already explored a state for that statement, stop.
    - The analysis depends only on the code and the current state
    - Continuing the analysis from this program point and state would yield the same results you got before
  - If the number of states isn't finite, too bad
    - Your analysis won't terminate

Example

```
1.  void foo(int x) {
2.      if (x == 0)
3.          bar(); cli();
4.      else
5.          baz(); cli();
6.      while (x > 0) {
7.          sti();
8.          do_work();
9.          cli();
10.     }
11.     sti();
12. }
```

Path 1 (before stmt): true/no loop
2: is_enabled
3: is_enabled
6: is_disabled
11: is_disabled
12: is_enabled

no errors

Example

```
1.  void foo(int x) {
2.      if (x == 0)
3.          bar(); cli();
4.      else
5.          baz(); cli();
6.      while (x > 0) {
7.          sti();
8.          do_work();
9.          cli();
10.     }
11.     sti();
12. }
```

Path 2 (before stmt): true/1 loop
2: is_enabled
3: is_enabled
6: is_disabled
7: is_disabled
8: is_enabled
9: is_enabled
11: is_disabled

already been here

Example

```
1.  void foo(int x) {
2.      if (x == 0)
3.          bar(); cli();
4.      else
5.          baz(); cli();
6.      while (x > 0) {
7.          sti();
8.          do_work();
9.          cli();
10.     }
11.     sti();
12. }
```

Path 3 (before stmt): true/2+ loops
2: is_enabled
3: is_enabled
6: is_disabled
7: is_disabled
8: is_enabled
9: is_enabled
6: is_disabled

already been here

Example

```
1.  void foo(int x) {
2.      if (x == 0)
3.          bar(); cli();
4.      else
5.          baz(); cli();
6.      while (x > 0) {
7.          sti();
8.          do_work();
9.          cli();
10.     }
11.     sti();
12. }
```

Path 4 (before stmt): false
2: is_enabled
5: is_enabled
6: is_disabled

already been here

all of state space has been explored

Example

```
1.  void foo(int x) {
2.      if (x == 0)
3.          bar(); cli();
4.      else
5.          baz(); cli();
6.      while (x > 0) {
7.          sti();
8.          do_work();
9.          cli();
10.     }
11.     sti();
12. }
```

- Analysis may be unsound
  - Race condition if bar() include an sti()

Assertion Side-Effects

```
{ #include <assert.h> }
// Apply SM ignoring control flow
sm Assert flow_insensitive {
  // Match expressions of "any" type
  decl { any } expr, x, y, z;
  // Used in combination to match all
  // calls with any arguments
  decl { any_call } any_fcall;
  decl { any_args } args;

  // Find all assert calls.  Then apply
  // SM to "expr" in state "in_assert."
  start: { assert(expr); } ==>
       { mgk_expr_recurse(expr, in_assert); } ;
  // Find all side-effects
  in_assert:
     // Match all calls
     { any_fcall(args) } ==>
           { err("function call"); }
      // Match any assignment (including
      // the operators +=, -=, etc.)
    | { x = y }  ==> { err("assignment"); }
      // Match all increments and decrements
      // --z and ++z ommited for brevity
    | { z++ } ==> { err("post-increment"); }
    | { z-- } ==> { err("post-decrement"); } ;
}
```

- Flags error if assert() has side effects
- Illustrates matching on sub-expressions
- 14 errors, 2 false positives in ExOS
- Example bug:
  - ExOS, mmap

```
/* libexos/os/mmap.c:mmap_fault_handler:410 */
assert(_exos_self_insert_pte(0, PG_P|
     PG_U|PG_W, PGROUNDDOWN(va), 0, NULL) == 0);
```

  - Causes VM fault if assert is disabled

Assertion Failure

- Perform reaching definitions *path-sensitively*
  - Different results for each path
- At an assert statement
  - If reaching definition of each variable in assert is a constant assignment, evaluate the assert
  - Flag an error if it is false
- Results
  - 5 errors in FLASH
  - Well-tested code
  - Def to assignment paths long and complex
    - e.g. 300 lines, 20 if statements, 4 else clauses, 10 conditional compilation directives

Tainted Analysis

- Kernel shouldn't trust data from user
  - Could pass null references
- Analysis
  - Assume all data from user initially in *tainted* state
  - Tainted data cannot be used except by functions that check its validity
  - 18 errors
  - 15 false positives
  - Example error:

```
/* from sys/kern/disk.c */
int sys_disk_request (u_int sn, struct Xn_name
         *xn_user, struct buf *reqbp, u_int k) {
  ...
  /* bypass for direct scsi commands */
  if (reqbp->b_flags & B_SCSICMD)
    return sys_disk_scsicmd (sn, k, reqbp);
```


Memory Management

- Similar to PREfix
  - Catch leaks, use after free, possible null dereferences
- Challenge: How to do this intra-procedurally?
  - It's common for procedures to return newly allocated memory
- Solution
  - Check error return paths
    - OS: those returning a negative integer
  - Catches many (but not all) errors
    - PREfix can do better using interprocedural analysis

Interprocedural analysis

- First, perform local analysis
  - e.g. does this function block?
  - e.g. are interrupts enabled?
- Later, perform reachability analysis on call graph
  - e.g. is a blocking function transitively called?
  - If so, interrupts better be enabled
- Can find only simple interprocedural errors
  - local analysis + reachability
- Vs. PREfix
  - Perform local analysis
  - Compute summary
  - Use summary to analyze callers
  - Handle recursion by exploring up to a fixed call depth
  - Can only track simple (language-level) information

Interprocedural analysis

- First, perform local analysis
  - e.g. does this function block?
  - e.g. are interrupts enabled?
- Later, perform reachability analysis on call graph
  - e.g. is a blocking function transitively called?
  - If so, interrupts better be enabled
- Can find only simple interprocedural errors
  - local analysis + reachability
- Vs. PREfix
  - Perform local analysis
  - Compute summary
  - Use summary to analyze callers
  - Handle recursion by exploring up to a fixed call depth
  - Can only track (language-level) information
- Vs. Fluid
  - Perform local analysis only
  - Use annotations to determine what a callee does
  - Can track sophisticated predicates but requires user input

Two Kinds of Path Sensitivity

- Metal
  - Explores all paths separately
  - Trims paths that share states at a program point
  - Does not keep track of predicates
- PREfix
  - Explores all paths separately
  - Keep track of predicates
  - Only explores feasible paths (based on predicates)

```
if (threads)
    lock(y);
do_something();
if (threads)
    unlock(y)
```

- Metal will report a double-unlock error
  - False positive!
- PREfix will not

Comparison

	Focus	Inter-procedural	Sound?
PREfix	Language errors	Summaries	No
Fluid	Concurrency errors	Annotations	Yes/ Contingent
Metal	Rule violations	Post-pass	No