A System and Language for Building System-Specific, Static Analyses

Seth Hallem, Benjamin Chelf, Yichen Xie, and Dawson Engler
Stanford University

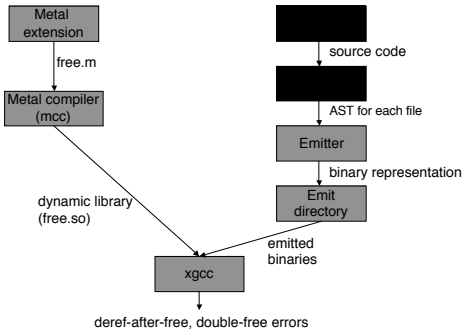
Overview

- Goal: find as many bugs as possible
  - Allow users of our system to write the analyses
- Implementation: tool with two parts
  - Metal - the language for writing analyses
  - xgcc - the engine for executing analyses
- System design goals
  - Metal must be easy to use and flexible
    - we have written over 50 checkers, found 1000+ bugs in Linux, OpenBSD and still counting
  - xgcc must execute Metal extensions efficiently
  - xgcc must not restrict Metal extensions *too* much

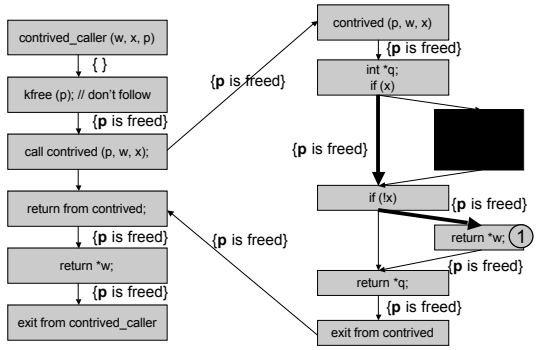
```
int contrived_caller (int *w, int x, int *p) {
    kfree (p);
    contrived (p, w, x);
    return *w;              // deref after free (3)
}

int contrived (int *p, int *w, int x) {
    int *q;
    if (x) {
        kfree (w);
        q = p;
        p = 0;
    }
    if (!x)
        return *w;  // safe (1)
    return *q;      // deref after free (2)
}
```

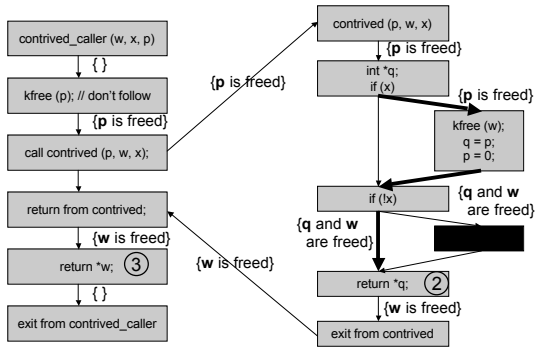
System Overview



Analysis Overview: if (!x) branch



Analysis Overview: if (x) branch

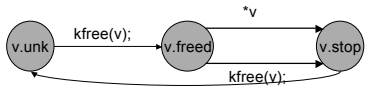


Metal extensions

- State machine abstraction
  - SMs have patterns, states, transitions, and actions
- Why is Metal easy to use?
  - SMs are a familiar concept to programmers
  - Patterns specify interesting source constructs in the source language
- Why is Metal flexible?
  - Actions are escapes to arbitrary C code that execute whenever a transition executes
  - Main restriction is determinism

Example: the free checker

- Looks for deref-after-free, double free
- Free checker is a collection of SMs
- Each SM tracks a single program object



–

Metal states

- Two types of states
  - Global: “interrupts are disabled”
  - Variable-specific: “pointer p is freed”
- States are bound to state variables

```
sm free-check {
        state decl any_pointer v;
        start: { kfree (v) } ==> v.freed;
        v.freed: { *v } ==> v.stop,
                { err (“dereferenced %s after free!”, mc_identifier (v)); }
        | { kfree (v) } ==> v.stop,
                { err (“double free of %s!”, mc_identifier (v)); }
        ;
}
```

Metal patterns

- Syntactic matching: literal AST match
- Semantic matching: wildcard types

```
sm free-check {
  state decl any_pointer v;
  start:    { kfree (v) } ==> v.freed;
  v.freed:{ *v } ==> v.stop,
            { err ("dereferenced %s after free!", mc_identifier (v)); }
          | { kfree (v) } ==> v.stop,
            { err ("double free of %s!", mc_identifier (v)); }
          ;
}
```

Metal transitions and actions

- Specify with source state, pattern, destination state
- Actions execute when transition occurs
  - Report errors, extend analysis (e.g., statistical)

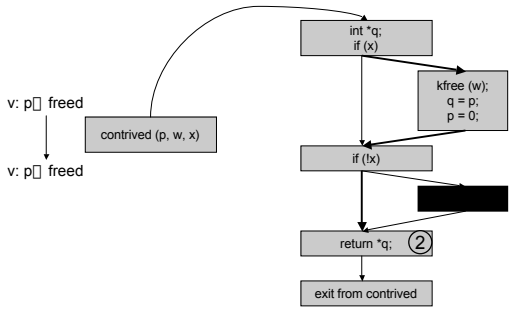
```
sm free-check {
  state decl any_pointer v;
  start: { kfree (v) } ==> v.freed;
  v.freed: { *v } ==> v.stop,
              { err ("dereferenced %s after free!", mc_identifier (v)); }
          | { kfree (v) } ==> v.stop,
              { err ("double free of %s!", mc_identifier (v)); }
          ;
}
```

Executing Metal SMs

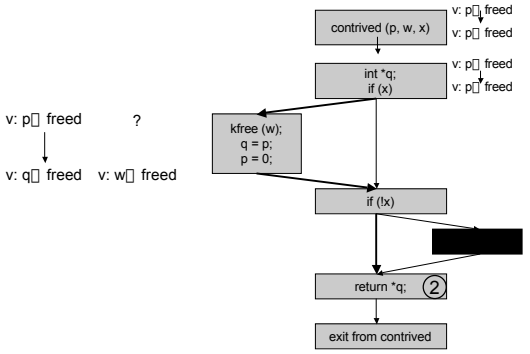
- Intraprocedural analysis:
  - Depth-first-search + caching
  - Cache at the block level
    - contains union of all "facts" seen at that block
  - On cache hit, abort the current path, backtrack
- Interprocedural analysis
  - Summarize the effects of analyzing large portions of the code
  - Use summaries whenever possible

Executing Metal SMs: DP Edges

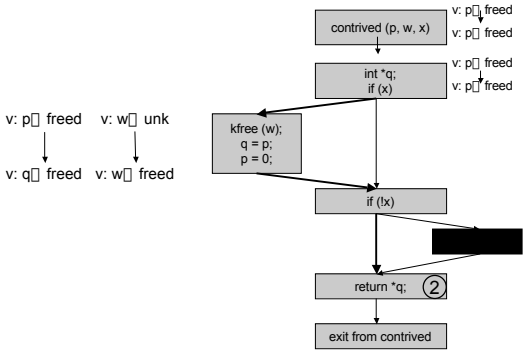
- Derived from "Precise Interprocedural Dataflow Analysis via Graph Reachability"; Reps, Horowitz, Sagiv 1995



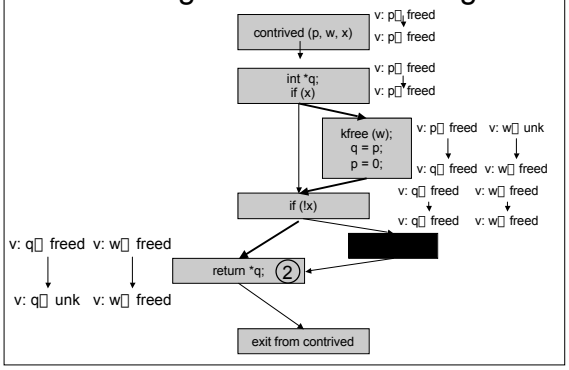
Executing Metal SMs: DP Edges



Executing Metal SMs: DP Edges



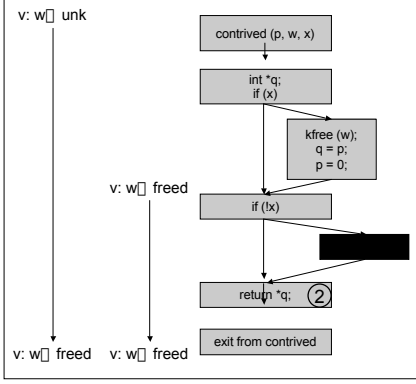
Executing Metal SMs: DP Edges



Memoizing Analysis Results

- Edges make statements of two forms
  - given that **p** is freed at the entry to block **b**, **p** will be freed at the exit from **b**
  - given that we know nothing about **w** at the entry to **b**, we know **w** is freed at the exit from **b**
- Relax edges
  - given that **p** is freed at the entry to block **b**, **p** will be freed at the exit from procedure **P**
- Use edges to transform extension state

Example: Memoizing Edges



Interprocedural Analysis

- Start at each entry point to the callgraph
  - initially we do not know any facts
- Traverse CFG for each function depth-first
- At the end of an intraprocedural path, relax edges
- At a function call, analyze call with new facts
- At return, apply edges to extension state

False-Path Pruning

```
int f (int x, int z) {          ← Know nothing.
        int a, b, p, q, y;
        p = x;                  ← Track p = x.
        q = 5;                  ← Track q = 5.
        a = x;                  ← Track a = x.
        b = 5;                  ← Track b = 5.
        if (z  == (p + q)) {    ← Track z = p + q.
                y = a + b;      ← Track y = a + b.
                if (z != y) {   ← ??
                        . . .
                }
        . . .
        }
}
```

False-Path Pruning

```
int f (int x, int z) {
        int a, b, p, q, y;
        p = x;                  ← {p, x}
        q = 5;                  ← {q, 5}
        a = x;                  ← {a, x}
        b = 5;                  ← {b, 5}
        if (z  == (p + q)) {    ← {z, p + q}
                y = a + b;      ← {y, a + b}
                if (z != y) {   ← ??
                        . . .
                }
        . . .
        }
}
```

False-Path Pruning

- Apply congruence closure

  {q, b, 5} {a, p, x} {z, y, a+b, p+q}
- Contradicts z != y

More False Positives

- Simple value flow
  - Tracks all value flow through direct assignment flow sensitively
  - Ignores indirect value flow
    - p = q implies p, q are aliases but not *p, *q
  - Tracks structure fields, pointer arithmetic

Unsoundness

- Unsound because:
  - No conservative alias analysis
  - Do not handle recursion soundly
- Benefits of unsoundness
  - Goal is to find as many bugs as possible
  - For many properties conservative assumptions cause an explosion of false positives
- Future goal: precise unsoundness

Conclusion

- Evaluating our approach
  - Flexible: over 50 checkers
  - Easy-to-use: Metal provides abstraction, sugar
    - unsound analysis is easy
  - Effective: 1000+ real bugs, still finding more
  - What makes our tool effective?
    - does just enough analysis to find bugs
    - often trade precision for speed/flexibility
    - aliasing: conservative is too imprecise; more aggressive analysis is helpful

More False-Path Pruning

- Redefine loop targets/do not prune in loops
- Predicate edges/cache entries
- Ignores aliasing
- Could benefit from abstract simulation insight

Ranking

- Ranking: we find too many errors to inspect
  - Rank most likely, easiest-to-diagnose errors first
  - Statistical ranking: use statistical test of significance to rank rules we check
    - reliable rules are usually followed

More metal patterns

- Wildcard types
  - Typed placeholders within a pattern

```
decl any_pointer p;
start: { kfree (p) } ==>
```

```
decl any_expr e;
start: { e++ } ==>
```

```
decl {int} i;
start: { i++ } ==>
```

- Callouts: query AST directly if necessary
- Composition: &&, ||

```
decl any_fn_call f;
decl any_args args;
{ f(args) } && ${ strstr (mc_identifier (f), "mem") } ==> …
```

- Restrictions on patterns
  - No star operator; must be meaningful in isolation

Interprocedural Analysis

- Start at each entry point to the callgraph
  - initially we do not know any facts
- Traverse CFG for each function depth-first
  - at each block, remove seen facts
- At the end of an intraprocedural path or a cache hit, relax edges and backtrack intraprocedurally
- At a function call, use the edges summarizing the call to transform the extension state
  - reanalyze the called function with any new facts
- After all paths in a function are analyzed, backtrack to return-site, apply edges to extension state at call

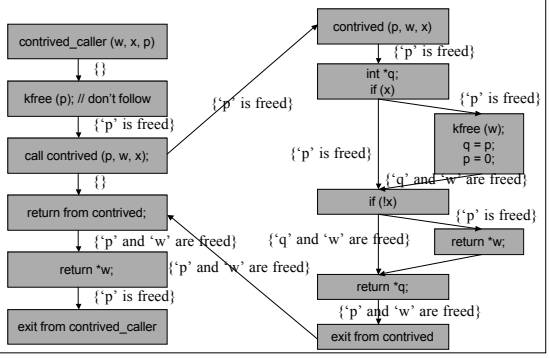
Overview

- The goal of our research is to find as many bugs in real systems as possible
- Insight: many rules are system-specific.
  - The number of rules that apply to all programs is very small; violations of these generic rules are hard to find.
    - E.g. memory errors, race conditions, etc.
- Programmers know the rules their code obeys
- A system that allows programmers to specify these rules will find lots of bugs

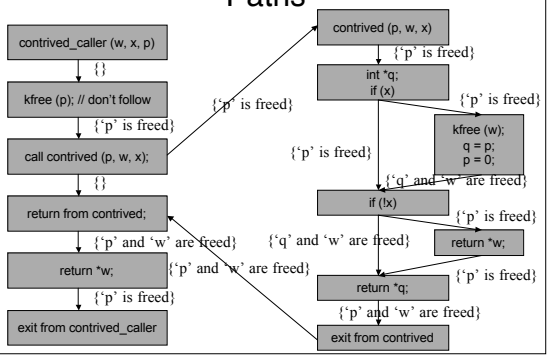
Executing Metal SMs

- Simple, slow method:
  - start at entry to main
  - follow each path through CFG
    - call extension on each AST node (program point)
    - if the current program point is a function call
      - retrieve CFG for called function
      - restart analysis at entry to callee
      - when analysis of callee finishes, return to appropriate return site
- Good attributes:
  - interprocedural, context-sensitive
- Unnecessary attributes:
  - path sensitive - SMs do not track data dependencies

Items (1) - (3): Overview



Analysis Result: Union of all Paths

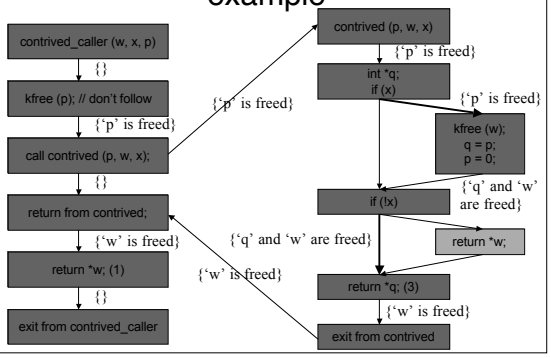


Metal actions

- Execute whenever a transition occurs
- Often used to print error messages, extend SM abstraction (e.g. add statistical analysis)

```
sm free-check {
  state decl any_pointer v;
  start: { kfree (v) } ==> v.freed;
  v.freed: { *v } ==> v.stop,
          { err ("using %s after free!", mc_identifier (v)); }
          | { kfree (v) } ==> v.stop,
          { err ("double free of %s!", mc_identifier (v)); }
          ;
}
```

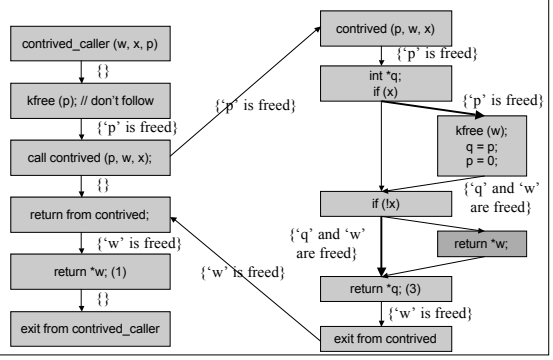
Executing Metal SMs: Free example



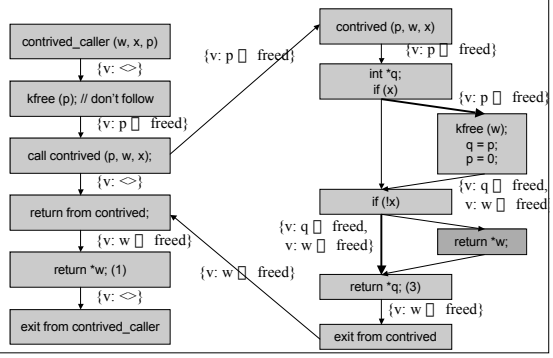
Interprocedural Analysis

- Start at each entry point to the callgraph
  - initially we do not know any facts
- Traverse CFG for each function depth-first
- At entry to block 'b', apply all relevant edges in 'b'
  - if there are no new facts, backtrack
- At the end of an intraprocedural path, backtrack
  - before each backtrack, relax edges
- At a function call, use the edges summarizing the call to transform the extension state
  - reanalyze the called function with any new facts
- After all paths in a function are analyzed, backtrack to return-site, apply new edges

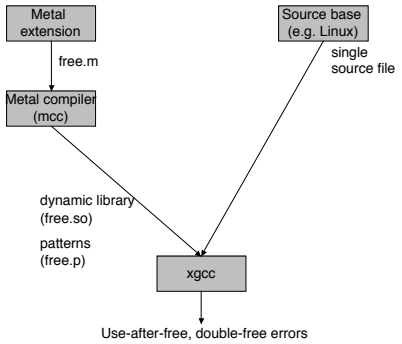
Analysis Overview: if (x) branch



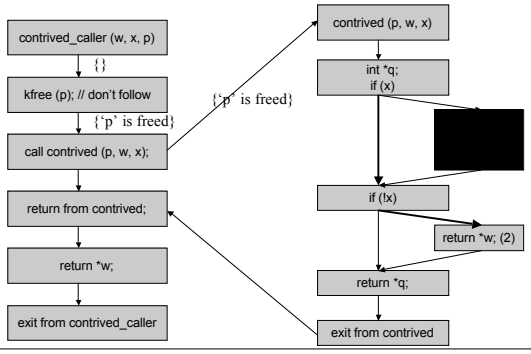
Executing Metal SMs



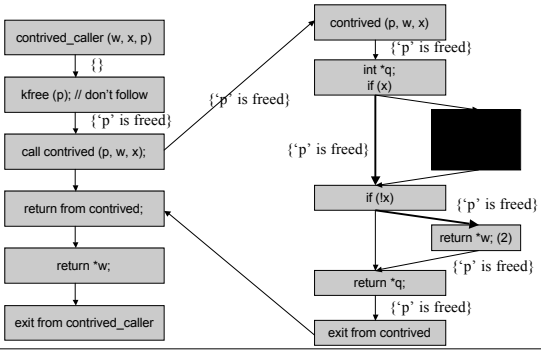
Old System



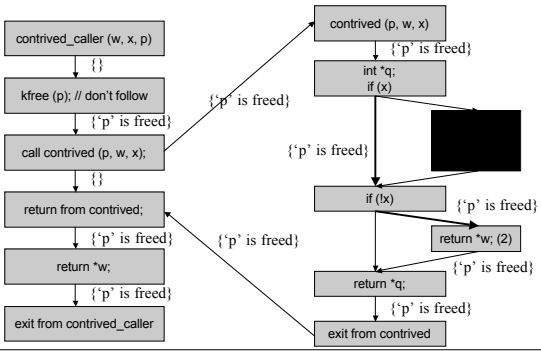
Analysis Overview: if (!x) branch



Analysis Overview: if (!x) branch



Analysis Overview: if (!x) branch



Handling False Positives

- False-path pruning
  - Intraprocedural only
  - Not conservative because it ignores aliasing
  - Predicate edges, cache entries
  - Track assignment list + conditional list
  - At each conditional branch
    - for each equality, place LHS, RHS in same equivalence class
    - use congruence closure to merge equivalence classes
    - look for a contradiction
  - To avoid loop unrolling, redefine loop targets, do not prune paths within loops

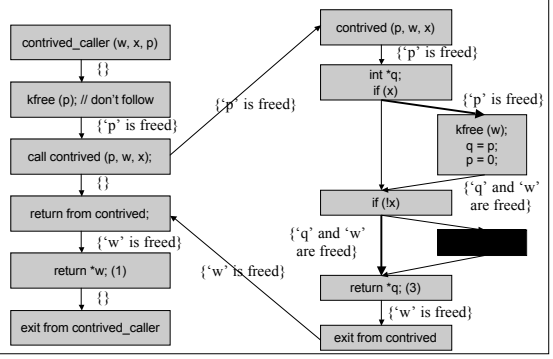
Example: Use-after-free errors

```
int contrived_caller (int *w, int x, int *p) {
    kfree (p);
    contrived (p, w, x);
    return *w;              //  (1) deref after free
}
```

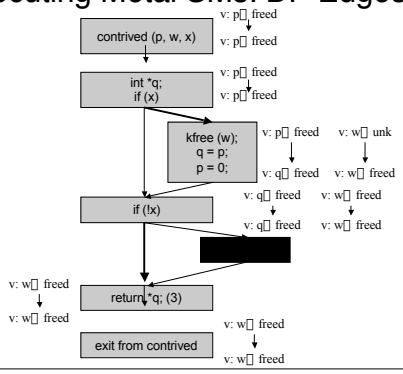
```
int contrived (int *p, int *w, int x) {
    int *q;
    if (x) {
        kfree (w);
        q = p;
        p = 0;
    }
    if (!x)
        return *w;  // (2) safe
    return *q;      // (3) deref after free
}
```

Output: “contrived: ERROR:FREE: Deref-after-free of ‘q’ at line 10 :: from kfree(p), contrived_caller line 2”

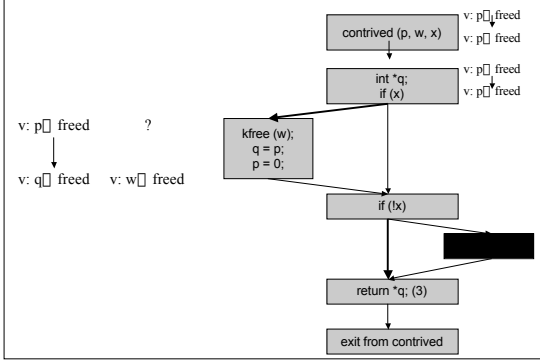
Analysis Overview: if (x) branch



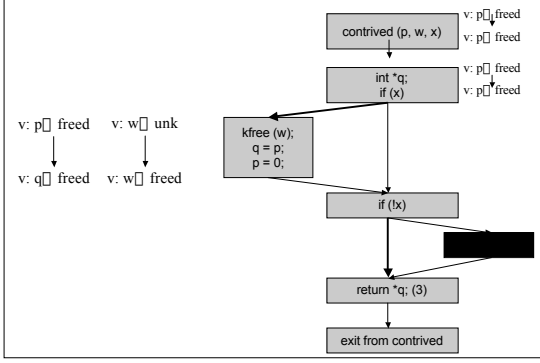
Executing Metal SMs: DP Edges



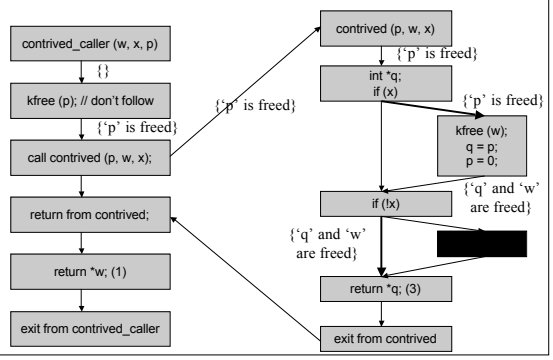
Executing Metal SMs: DP Edges



Executing Metal SMs: DP Edges



Analysis Overview: if (x) branch



Example: the free checker

- Looks for use-after-free, double free
- Free checker is a collection of SMs
- Each SM tracks a single program object

```
sm free-check {
  state decl any_pointer v;
  start: { kfree (v) } ==> v.freed;
  v.freed: { *v } ==> v.stop,
           { err ("using %s after free!", mc_identifier (v)); }
           | { kfree (v) } ==> v.stop,
           { err ("double free of %s!", mc_identifier (v)); }
           ;
}
```