Systems and Internet Infrastructure Security

Network and Security Research Center
Department of Computer Science and Engineering
Pennsylvania State University, University Park PA

Static Analysis Basics II

Trent Jaeger
Systems and Internet Infrastructure Security (SIIS) Lab
Computer Science and Engineering Department
Pennsylvania State University

September 19, 2011

Outline

- More background
  - Pushdown Systems
  - Boolean Programs
  - Enable more refined dataflow analysis
- Metacompilation
- Control Flow and Data Flow Integrity

Pushdown Systems

- To encode ICFGs
  - What are ICFGs?
  - Why are they necessary for dataflow analysis?
  - What is the major challenge in using ICFGs in dataflow?
  - Other challenges?

Pushdown Systems

- Consists of
  - A finite set of states
  - A finite set of stack symbols
  - A finite set of rules
    - Which define a transition relation

Modeling Control Flow

- One state
- Each ICFG node is a stack symbol
- Each ICFG edge is represented by a rule
  - $(p, e_{main}) \rightarrow (p, n_1)$
  - $(p, n_3) \rightarrow (p, e_f n_4)$
  - $(p, n_{12}) \rightarrow (p, x_f)$
  - $(p, x_f) \rightarrow (p, epsilon)$
- PDSs with a single control location are called context-free processes

Pushdown Systems

- A configuration is a pair (node, stack)
  - Where we are currently and why
  - Pre and post-configurations are important
    - Backward and forward reachability over the transition relation

Find All Reachable Configurations

- Start with a set of configurations
  - Can be used for assertion checking statically (Phil)
- Number of configurations in a pushdown system is unbounded – use finite automata to describe regular sets of configurations
- Why?
  - Symbolic Reachability Analysis of Higher-Order Context-Free Processes – Bouajjani and Meyer
  - http://igm.univ-mlv.fr/~ameyer/binaires/fsttcs04.pdf

Find All Reachable Configurations

- Represent sets of configurations as
- P-automaton (FSA)
  - States (superset of PDS states)
  - Stack symbols
  - Transition relation
  - Start and final states
- What is it missing from the PDS representation?

Find All Reachable Configurations

- Compute post*(C) and pre*(C)
- Take a P-automaton that accepts a set of configurations C
  - Produces an automaton that accepts the pre and post configurations
- Saturation procedures
  - Add transitions to A until no more can be satisfied

Find All Reachable Configurations

- Prestar
  - If *(p, v)* → *(p', w)* and *p'* $\rightarrow_w$ *q* in *A*
    - *v in Stack, w in Stack**
  - Then add transition *(p, v, q)*
- Why does this enable finding the backward reachable state for a configuration?
  - Efficient algorithms for modeling pushdown systems, Esparza et al (ref 107)

Find All Reachable Configurations

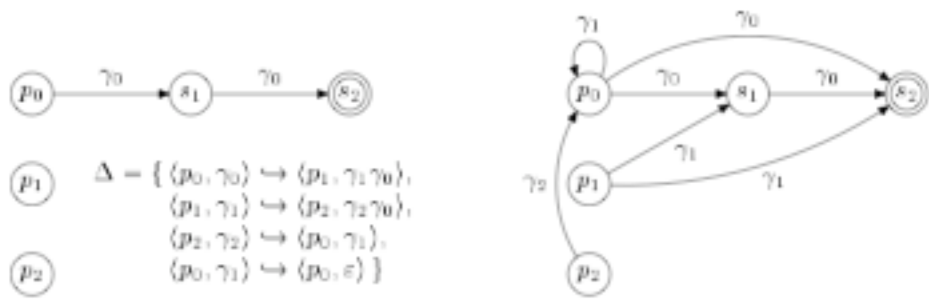


Fig. 1. The automata $\mathcal{A}$ (left) and $\mathcal{A}_{pre^*}$ (right)

Find All Reachable Configurations

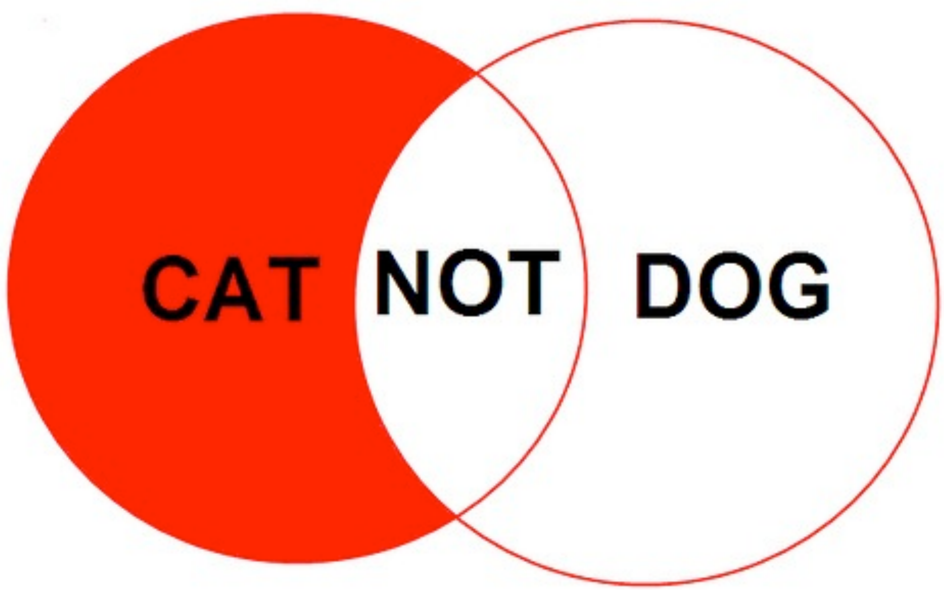
- Poststar
  - Phase I: For each (p', v') s.t. P contains at least one rule $(p, v) \rightarrow (p', v', v'')$, add new state $p'_{v'}$
  - Phase II:
    - If $(p, v) \rightarrow (p', epsilon)$ in rules $p \rightarrow_v q$, then $(p', epsilon, q)$
    - If $(p, v) \rightarrow (p', v')$ in rules $p \rightarrow_v q$, then (p', v', q)
    - If $(p, v) \rightarrow (p', v'v'')$ in rules $p \rightarrow_v q$, then (p', v', pv') and $(p'_{v'}, v'', q)$
- Figure 2.7

Find All Reachable Configurations

- Fig 2.7
- Phase 1: Add states
  - $(p, n_3) \rightarrow (p, e_f n_4)$ results in P_{ef}
  - (p, n_7) also – but same state
- Phase 2: Add transitions
  - $(p, x_f) \rightarrow (p, epsilon) \rightarrow (p, epsilon, p_{ef})$ and $(p, epsilon, q)$
  - $(p, n_8) \rightarrow (p, n_9) \rightarrow (p, n_9, q)$
  - $(p, n_3) \rightarrow (p, e_f n_4)$ and $p \rightarrow q$, $\rightarrow (p, e_f, p_{ef})$ and (p, n_4, q)

Boolean Programs

- Program that only uses boolean data types and fixed-length vectors of booleans
  - Finite set of globals and local variables



Boolean Programs

- Let *G* be the valuations of globals
- Val_i be the valuations of the locals in procedure *i*
- L is local states
  - Program counter
  - Val_i
  - Stack
- Assignment statement is binary relation that states how the values *G* and Val_i (variables in scope) may change

Encode Boolean Program in PDS

- Why?
- Changes
  - Use P to encode globals
  - Use stack alphabet to encode local vars
- Model
  - (N_i is control nodes in i^{th} procedure)
  - P is set to G
  - Stack symbols are union of N_i X Val_i
  - Rules for assignments, calls, returns

Vulnerability

- How do you define computer ‘vulnerability’?
  - *Flaw*
  - *Accessible to adversary*
  - *Adversary has ability to exploit*

Vulnerability

- How do you define computer 'vulnerability'?
  - *Flaw – Can we find flaws in source code?*
  - *Accessible to adversary – Can we find what is accessible?*
  - *Adversary has ability to exploit – Can we find how to exploit?*

Bugs

- Known incorrect functions
  - Dereference after free
  - Double free
- Often have known patterns
  - Can we express and check

Metacompilation

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

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

Stanford University

Metacompilation

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

Metacompilation

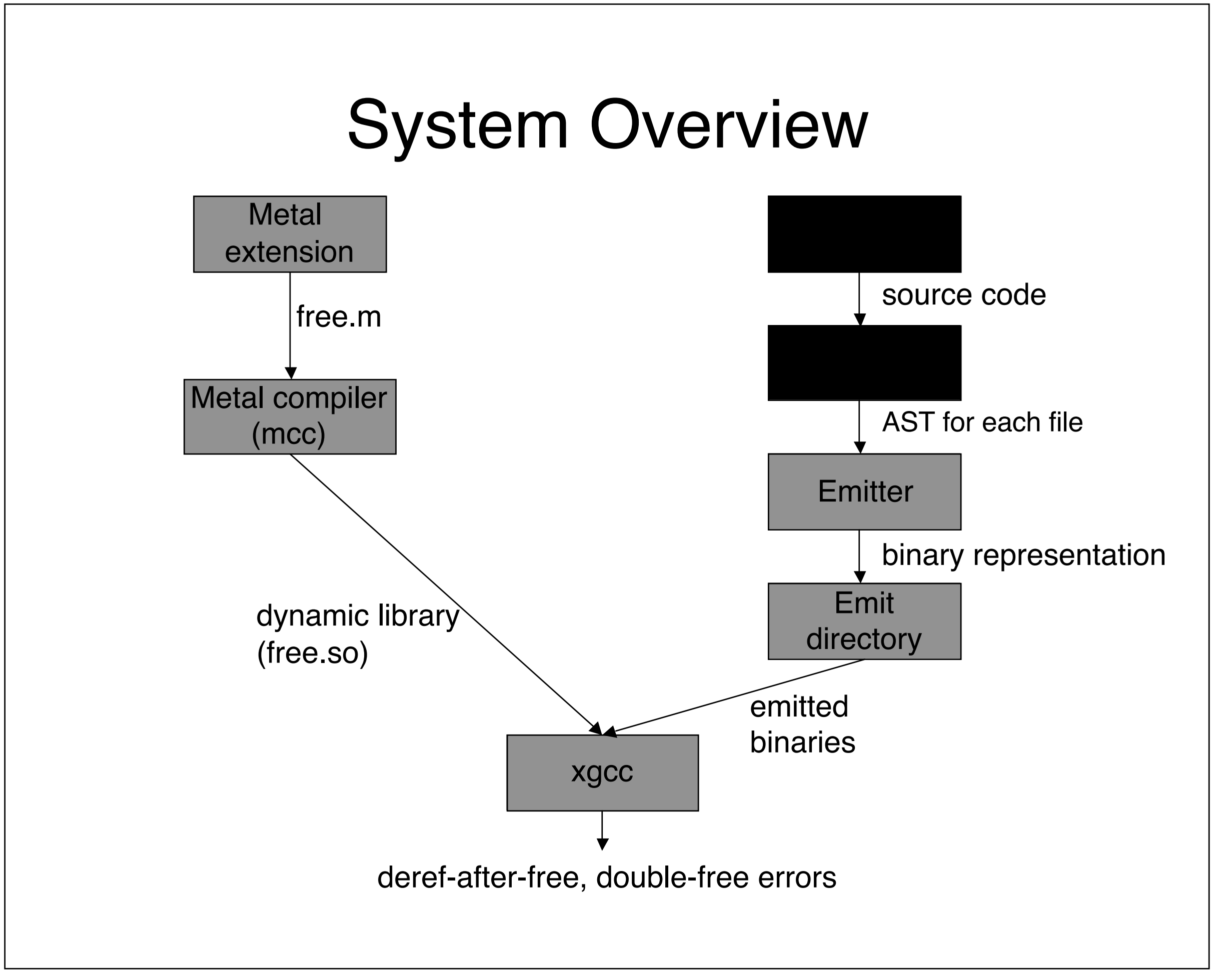
Overview

- 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

Metacompilation

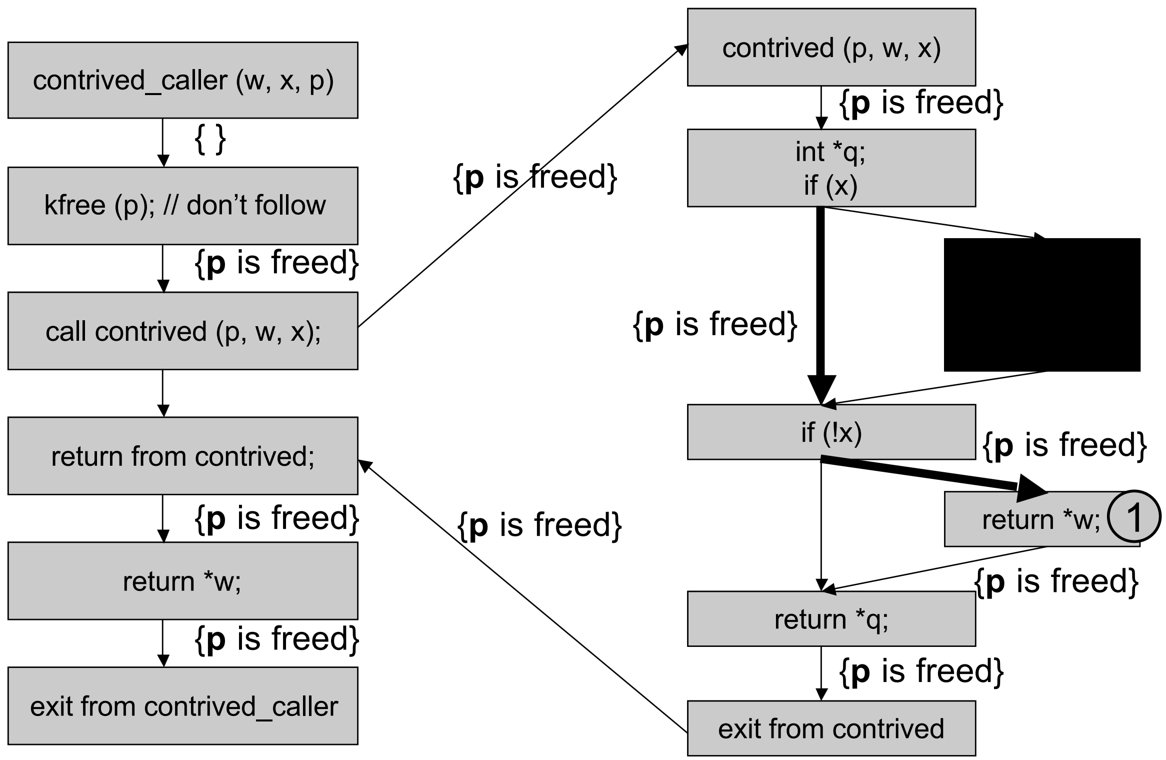
```
int contrived_caller (int *w, int x, int *p) {
    kfree (p);
    contrived (p, w, x);
    return *w;              // deref after free (3)
}

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)
}
```

Metacompilation

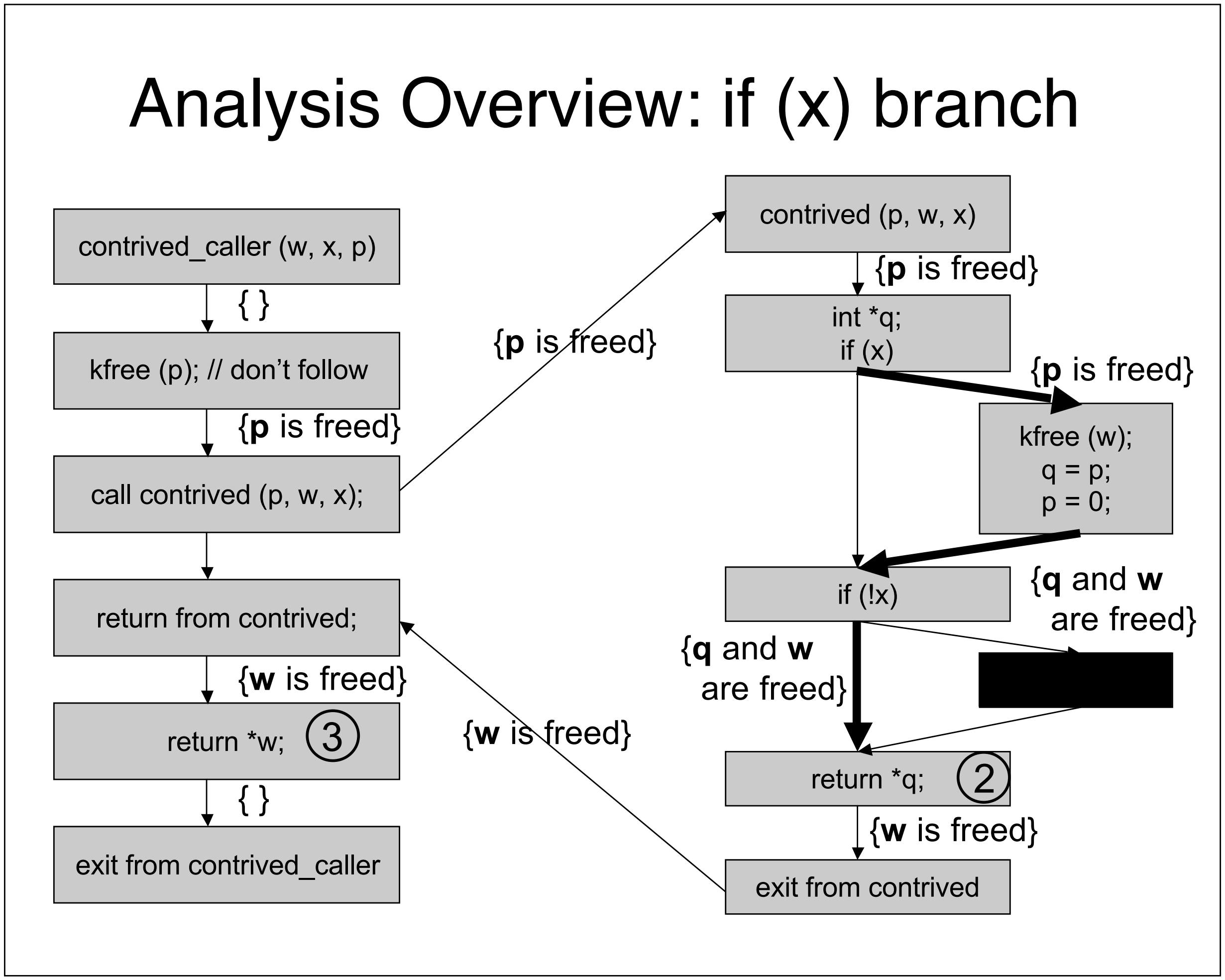


Metacompilation

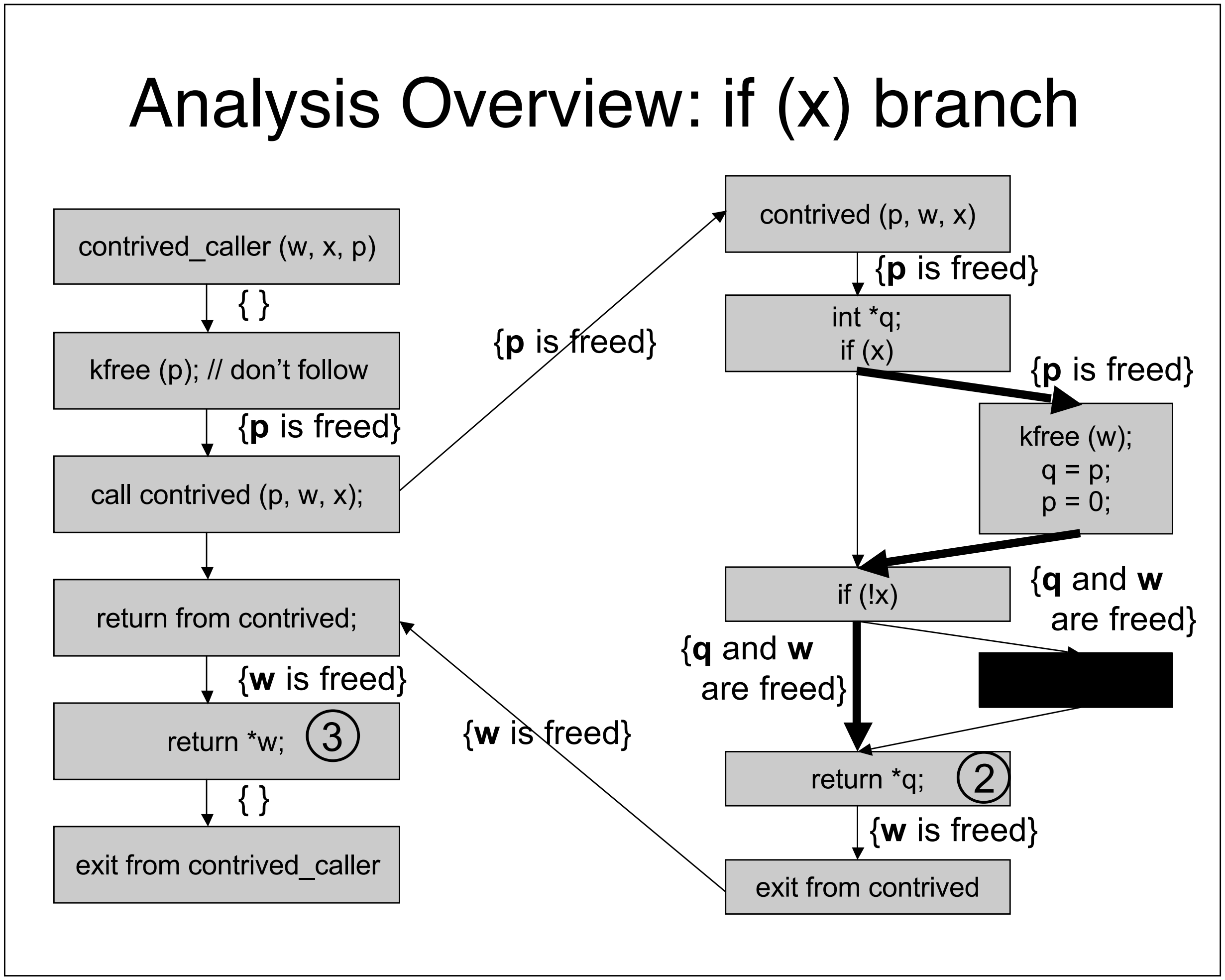
Analysis Overview: if (!x) branch



Metacompilation



Metacompilation



Metacompilation

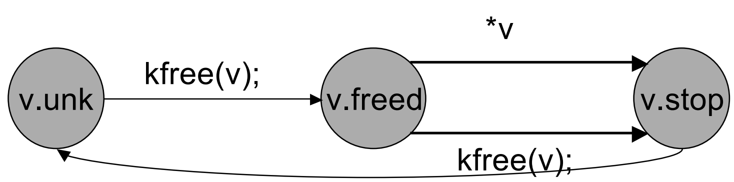
Metal extensions

- 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

Metacompilation

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



–

Metacompilation

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)); }
        ;
}
```

Metacompilation

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)); }
           ;
}
```

Metacompilation

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)); }
           ;
}
```

Metacompilation

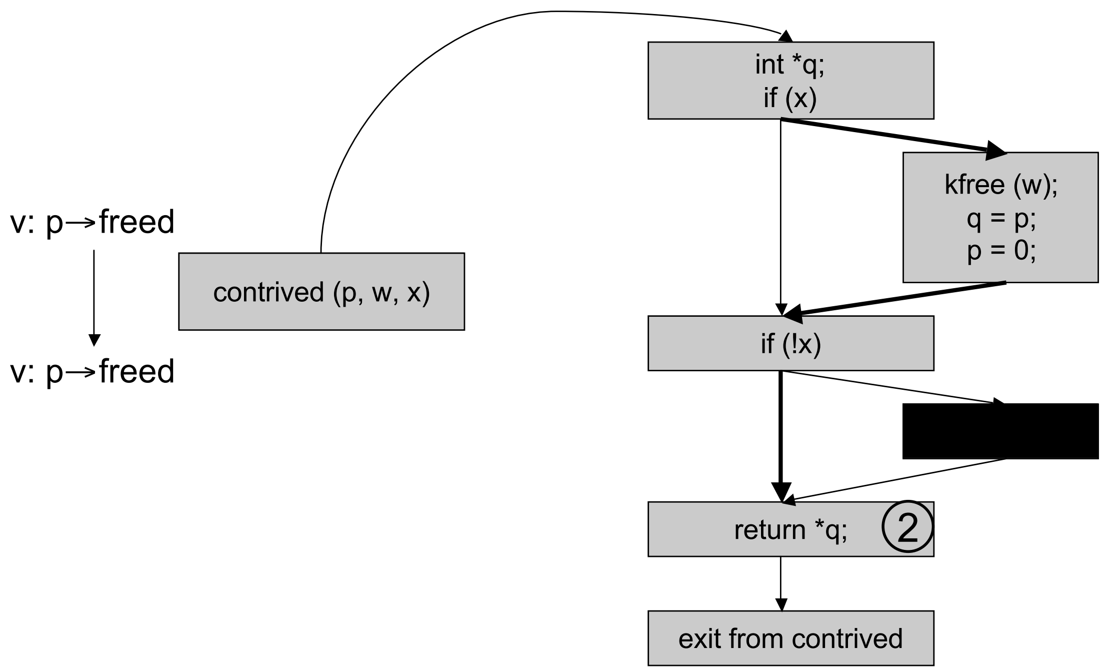
Executing Metal SMs

- 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

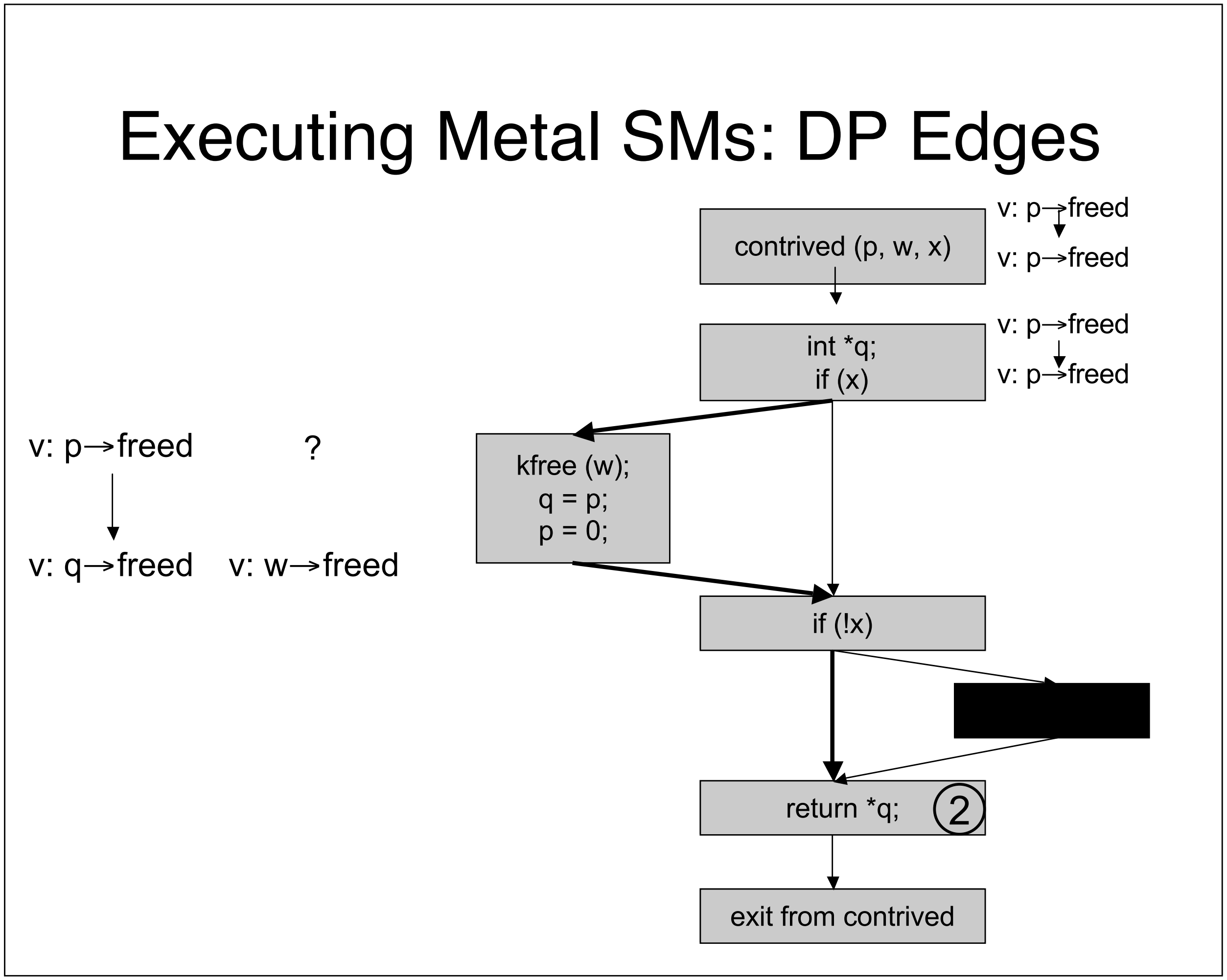
Metacompilation

Executing Metal SMs: DP Edges

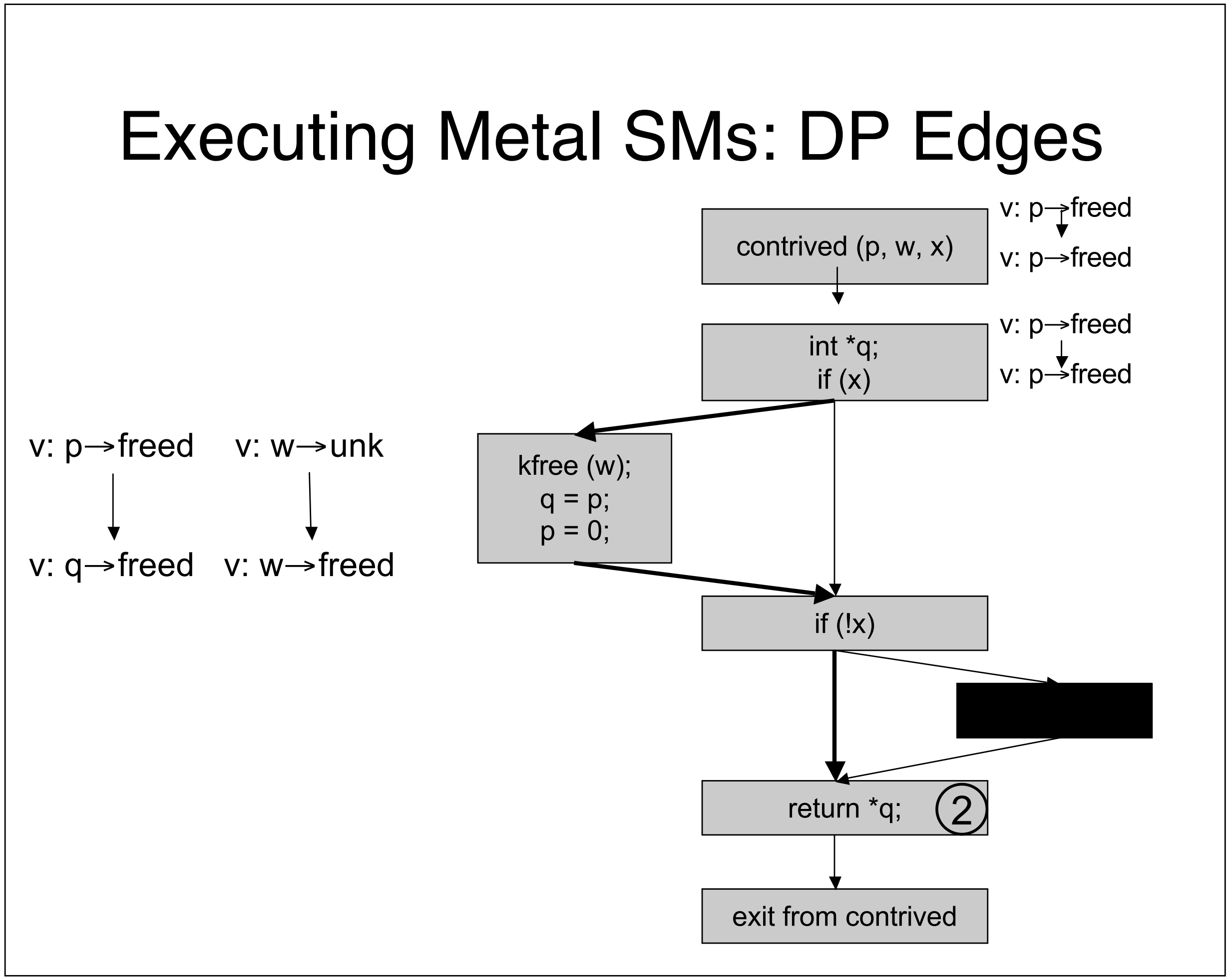
- Derived from “Precise Interprocedural Dataflow Analysis via Graph Reachability”; Reps, Horowitz, Sagiv 1995



Metacompilation

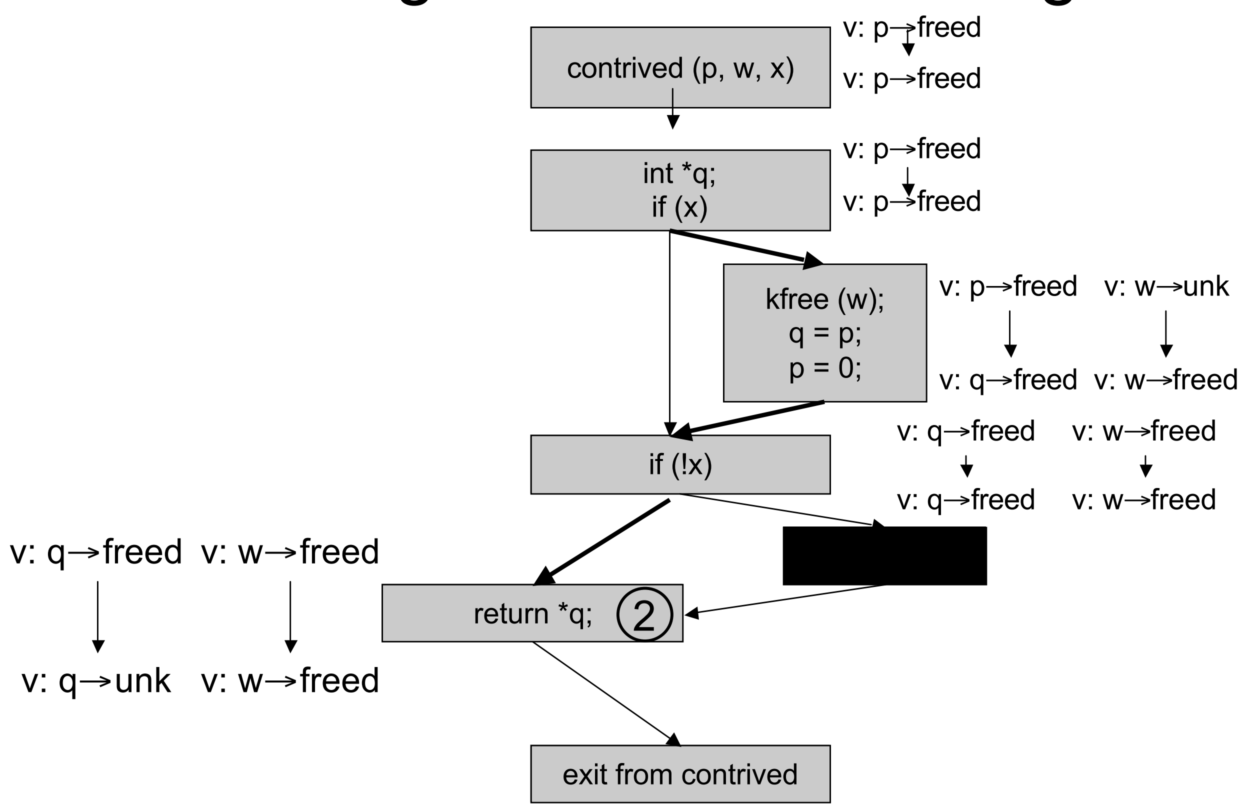


Metacompilation



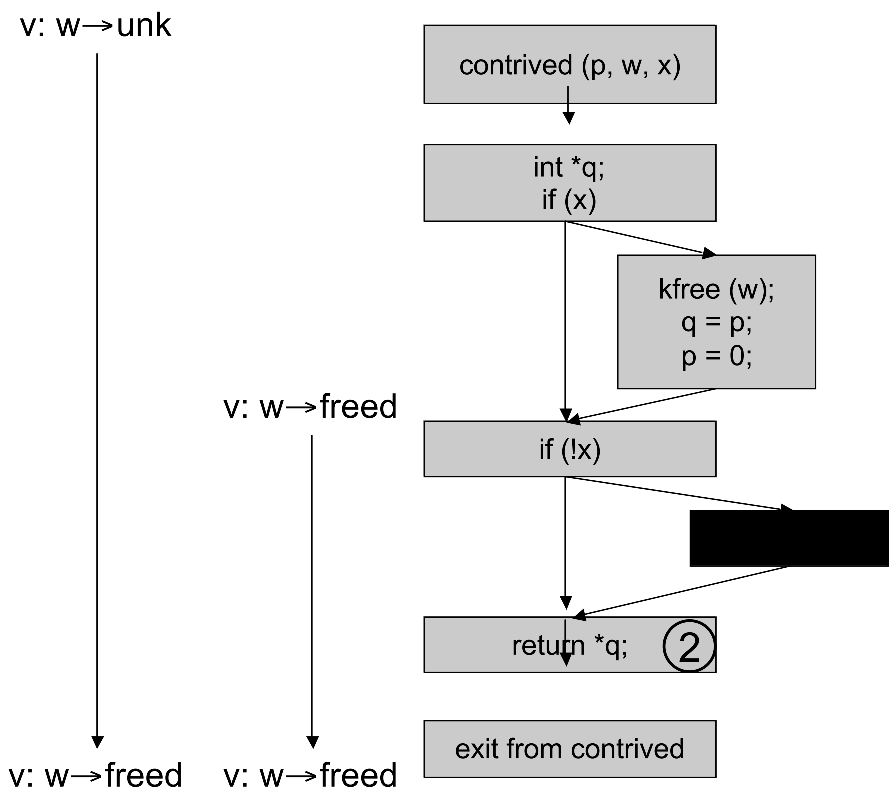
Metacompilation

Executing Metal SMs: DP Edges

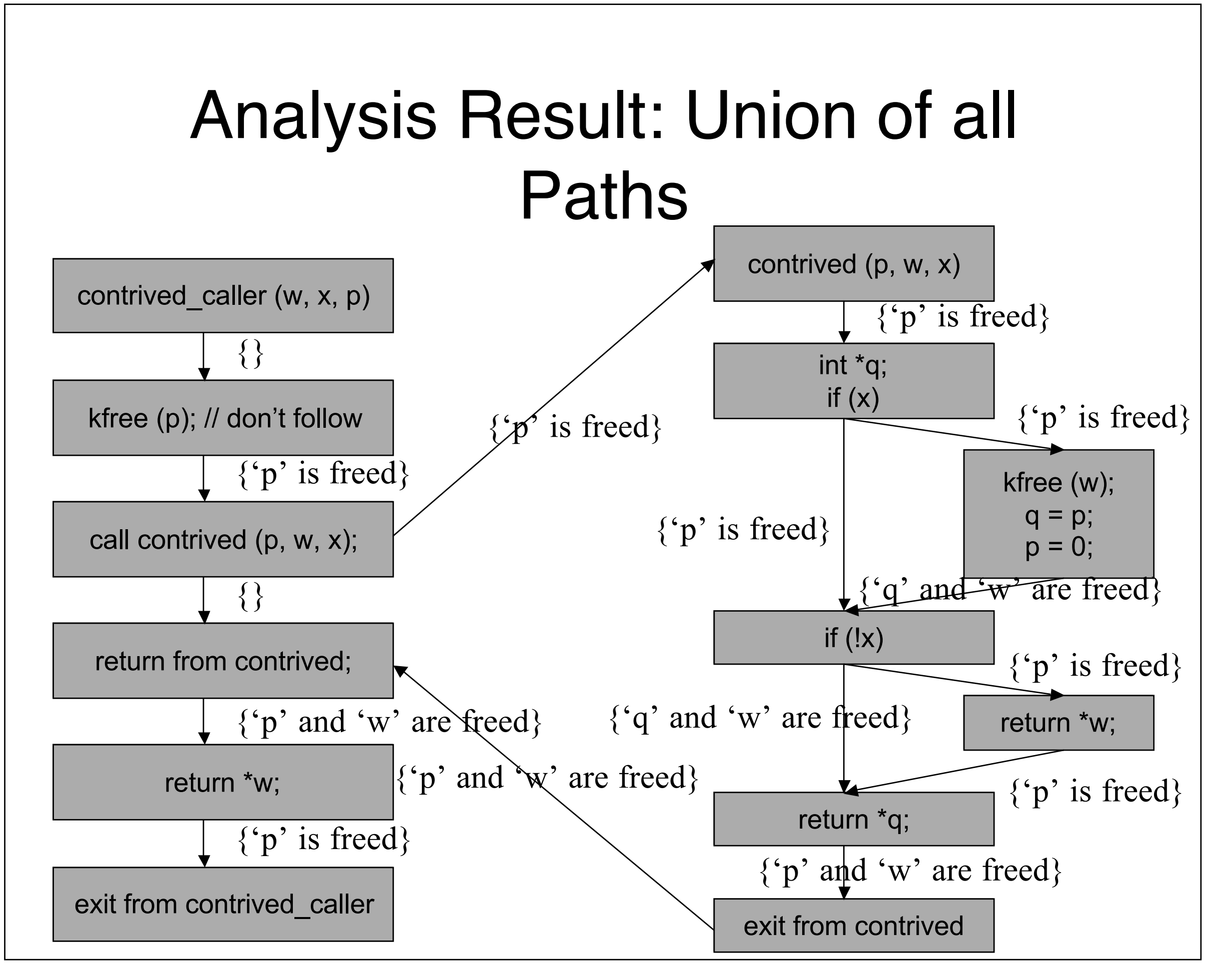


Metacompilation

Example: Memoizing Edges



Metacompilation



Metacompilation

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

Metacompilation

False-Path Pruning

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

- Know nothing.
- Track p = x.
- Track q = 5.
- Track a = x.
- Track b = 5.
- Track z = p + q.
- Track y = a + b.
- ??

Metacompilation

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)) {
                y = a + b;     <----------  {z, p + q}
                if (z != y) {  <----------  {y, a + b}
                        . . .  <----------  ??
                }
        . . .
        }
}
```

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

Metacompilation

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

Metacompilation

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

Metacompilation

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

Control and Data Flow Integrity

- How do they work?
- Are they Sound?

Summary

- Introduction to Pushdown Systems and Boolean Programs
  - Application to Dataflow Analysis
  - Prove to yourself
- Application of Static Analysis to Bug Finding
  - Metacompilation
- And Enforcement of Program Execution Integrity
  - Control Flow Integrity
  - Data Flow Integrity