Code: analysis, bugs, and security

supported by Bitdefender

Static analysis

Marius Minea

marius@cs.upt.ro

8 November 2017

Static analysis

Derive information about what a program does
without executing the program

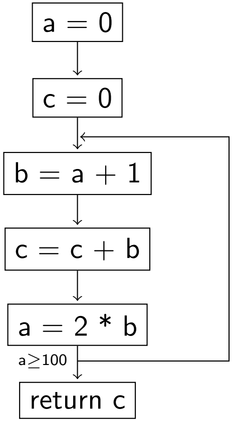
Desired info varies widely:
- what values does it compute? (range, overflow?)
- how many registers are needed ? (compiler)
- how much time? (worst-case execution time - WCET)
- does it reach an error state?

Complexity - precision tradeoff

Some problems are undecidable (Turing halting problem)

Analysis: done on CFG (control flow graph)

```
int a = 0, b, c = 0;
do {
  b = a + 1;
  c = c + b;
  a = 2 * b;
} while (a < 100);
return c;
```

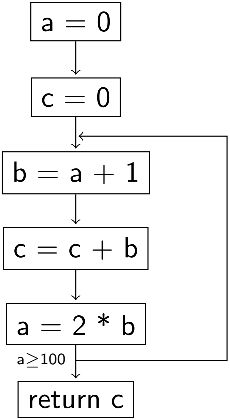


nodes are *basic blocks*: straight-line code segments with single entry and exit

Dataflow analyses

Calculate possible set of *values* at various program points.

```
1 int a = 0, b, c = 0;
2 do {
3   b = a + 1;
4   c = c + b;
5   a = 2 * b;
6 } while (a < 100);
7 return c;
```



What *value* means, depends on problem.

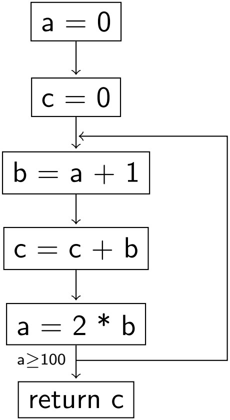
reaching definitions: where could variable have been last assigned?

(where does the value come from)?

line 3: a last assigned at line 1 or 5

Dataflow analyses: Live variables

```
1  int a = 0, b, c = 0;
2  do {
3    b = a + 1;
4    c = c + b;
5    a = 2 * b;
6  } while (a < 100);
7  return c;
```



live variables: might the value still be needed in the future ?

(do we still need a register for it?)

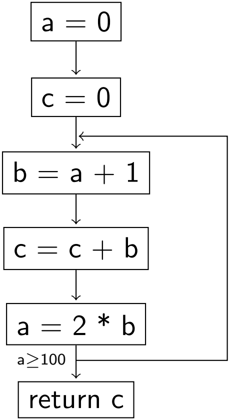
at (i.e., before) line 3: still need c and a, not b (gets assigned)

line 4 and 5: need c and b

line 6: need a and c

Dataflow analyses: Live variables

```
1  int a = 0, b, c = 0;
2  do {
3    b = a + 1;
4    c = c + b;
5    a = 2 * b;
6  } while (a < 100);
7  return c;
```



live variables: might the value still be needed in the future ?

(do we still need a register for it?)

at (i.e., before) line 3: still need c and a, not b (gets assigned)

line 4 and 5: need c and b

line 6: need a and c

How do we compute these values?

Must traverse CFG. How many times ?

Imperfect analogy: shortest paths in graph (all-pairs)

```
for (k = 0; k < n; ++k)
  for (i = 0; i < n; ++i)
    for (j = 0; j < n; ++j)
      if (d[i][k] + d[k][j] < d[i][j])
        d[i][j] = d[i][k] + d[k][j]
```

Relevant points:

compute some values over entire graph (here: node pairs)

runs *while a change propagates* (shorter path found)

$\Rightarrow$ stops when *transformation produces no change*

No change: $f(x) = x$ *fixpoint* of applied transformation

Worklist algorithm

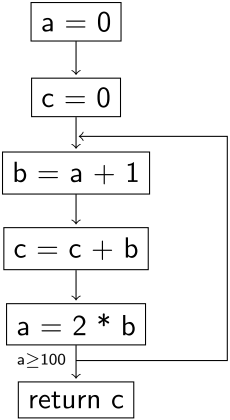
foreach s **do** $Out(s) = \top$ // no info
$W = \{entry\}$ // worklist
do
  choose $s \in W$ // statement to be considered
  $W = W \setminus \{s\}$ // remove from worklist
  old $= Out(s)$ // save current value of interest
  $In(s) =$ join $Out(s')$ forall $s' \in pred(s)$ // update inputs
  $Out(s) = Transfer_s(In(s))$ // apply meaning of statement
  if $Out(s) \neq old$ **then** // recompute affected successors
    forall $s' \in succ(s)$ **do** $W = W \cup \{s'\}$
while $W \neq \emptyset$

Values at each statement are *sets*. If universe of values *finite*, and computed functions are *monotone*, worklist algorithm terminates

Often: sets of boolean properties $\Rightarrow$ *bitvector frameworks*
  variable v_k live at line l ? def. at line i reaches line j ?

Dataflow analyses: Value analysis

```
int a = 0, b, c = 0;
do {
  b = a + 1;
  c = c + b;
  a = 2 * b;
} while (a < 100);
return c;
```



In general, set of values not finite. May approximate with *intervals*

Can easily derive $a \leq 99$, $b \leq 100$, $c \leq$???

depends on sophistication of math theories and abstract domains

Analyses: flow-sensitive or flow-insensitive

Does the analysis consider/keep track of statement order ?

Don't need for some analyses:
e.g., compute *call graph*

Indispensable for some others
anything involving *dependencies*

Makes a big difference in complexity
e.g. for pointer / *alias analyses*
(computing *points-to* sets)

Analyses: intra- vs. interprocedural

intraprocedural
analyze each function individually
cannot keep of constraints due to sequence of calls, values passed

interprocedural
whole-program analyses
complex, need to keep track of control and data flow between functions

match calls and returns to avoid spurious paths
  $k-$CFA: keep track of last k calls (*call strings*)
interprocedural *taint analysis*: a graph sources-sinks problem

Analyses: path-sensitive or path-insensitive

Example: [Das, Lerner, Seigle, PLDI 2002]

```
1   int main() {
2     if (dump)
3       f = fopen(dumpFile, "w");
4     if (p) x = 0;
5     else x = 1; // irrelevant for f
6     if (dump)
7       fclose(f);
8   }
```

If we merge info after every if, correlation is lost

At (5), `f` could be uninit or open

(no correlation with `dump`)

⇒ spurious warning

Computing function summaries

```
int f(int x, int y, int z)
{
  int r;
  if (x > 0) {
    r = y + 2 * x;
  } else {
    r = x + 3 * z;
  }
  return r;
}
```

Establish relations between inputs and outputs
  forward computation

  or backward computation
    (what values could have produced current result?)

need to deal with loops:
assume bounds on loop iterations, or try to compute fixpoints

Analyses: context-sensitive or context-insensitive

context-insensitive
function summary computed once, independently of call site

context-sensitive
function summary specialized for each call site

tradeoff precision for complexity

Small checkers are great

Engler, Chelf, Chou, Hallem: Checking System Rules Using System-Specific, Programmer-Written Compiler Extensions, OSDI 2000 (best paper)

went on to build Coverity

many other papers on simple, small, efficient static checkers
- for mining error patterns
- for concurrent programs
- etc.