Static Program Analysis

Xiangyu Zhang
What is static analysis

- Static analysis analyzes a program without executing it.
- Static analysis is widely used in bug finding, vulnerability detection, property checking
  - Easier to apply compared to dynamic analysis (as long as you have code)
    - The user does not even need to know how to run it
  - Better scalability compared to some dynamic analysis (e.g. tracing)
  - Findbug, coverity, codesurfer
Two kinds of Static Analysis

- **Syntax/structure oriented analysis**
  - They don’t try to understand the semantics of a program. Instead, they look at syntax and structure of a program
    - CFG, dominator, post-dominator, loop detection
  - A lot of applications
    - Code clone detection (text comparison, AST comparison, CFG comparison)
    - Malware analysis
    - Serve as the foundation for other advanced static/dynamic analysis
  - Limitation: cannot reason about program semantics and program state

- **Semantics oriented analysis (our focus)**
Lets start with the Simplest Static Analysis

What are the possible definitions for each use

1    z=…
2    x=…
3   if (…)  
4      x=…
5  else
6      s1
7    z=…
8   if (…)  
9      y=…x…
10  else
11     y=…z…
What are the possible call targets

```c
1  p=F1   /*F1, F2, F3, F4, F5 are functions*/
2  q=F2
3  x=input ()
4  if (...) 
5     q=F3
6  else
7     p=F4
8  if (...) 
9     p=F5
10 else
11    p=q;
12 (*p) (...) 
```
What is the range of possible values for a integer var.

1   x=10
2   y=input()
3   i=x+y
4   if (i>20)
5       i=20
6   else
7       z=input()
8   if (3<z<5)
9       i=i-z
10  print z
The first ingredient of static analysis

- Abstract domain
  - The results we want to compute by static analysis

- Transfer function
  - How the abstract values are computed/updated at each relevant instruction
    - Need to consider the instruction semantics
What are the possible call targets

1 x=F1     /*F1, F2, and F3 are functions*/
2 y=F2
3 q=&x
4 if (...) 
5 x=F3
6 else
7 p=&x
8 if (...) 
9 p=q
10 else
11 p=&y;
12 *(*p) (...)

What about loops

When shall we terminate a loop path?

- Analyze the possible sign of a variable

```
1 x=input()
2 while (...)
3   x=-x
```

Since we are always interested in the aggregation of abstract values along all paths. If the aggregation stabilizes, we shall terminate

- Monotonically growth
- The abstract domain is finite
A semi-lattice is a domain of values $V$ and a meet operator $\land$ such that,

- $\forall a, b, \& c \in V$:
  1. $a \land a = a$ (idempotent)
  2. $a \land b = b \land a$ (commutative)
  3. $a \land (b \land c) = (a \land b) \land c$ (associative)

- $\land$ imposes a partial order on $V$, $\forall a, b, \& c \in V$:
  1. $a \geq b \iff a \land b = b$
  2. $a > b \iff a \geq b$ and $a \neq b$
  3. $a \geq b$ and $b \geq c$, then $a \geq c$

- A semi-lattice has a top element, denoted $T$
  1. $\forall a \in V, a \leq T$
  2. $\forall a \in V, T \land a = a$
Semi-lattices for previous examples

- Def[\(x@n\)]: the possible definitions of \(x\) at \(n\)

\[
\begin{align*}
\{\} & \quad \top \\
\{d_1\} & \quad \{d_2\} & \quad \{d_3\} \\
\{d_1, d_2\} & \quad \{d_1, d_3\} & \quad \{d_2, d_3\} \\
\{d_1, d_2, d_3\} & \quad \bot
\end{align*}
\]
Lattice + monotonicity + finite height = termination

Are we there yet?

- Path explosion, e.g. a program with n diamonds.
Avoid Analyzing Individual Paths

- Analyze multiple paths at a time and compute aggregate information directly.

- $\text{Def}_{\text{in}}[x@n]$: all the possible definitions of $x$ along some path reaching $n$ (before getting through $n$)
  \[ \text{Def}_{\text{in}}[x@n] = \bigwedge_{n's\ predecessor\ n_p} \text{Def}_{\text{out}}[x@n] \]

- For any $x! = y$ (node $n$ is “$y = ...$”)
  \[ \text{Def}_{\text{out}}[x@n] = \text{Def}_{\text{in}}[x@n] \]

- $\text{Def}_{\text{out}}[y@n] = \{n\}$
Other Examples

- Call target analysis
- Range analysis
Worklist Algorithm

For each block node $n$ and every variable $x$
$\text{AD}^\text{in}[x@n] = \text{Ad}_{\text{out}}[x@n] = \emptyset$
change = true;
while change do begin
  change = false;
  for any $n$ and $x$
    $\text{AD}^\text{in}[x@n] = \bigwedge_{n's\ predecessorn_p} \text{AD}_{\text{out}}[x@n]$
    oldvalue = $\text{Ad}_{\text{out}}[x@n]$;
    $\text{Ad}_{\text{out}}[x@n] = F(\text{AD}^\text{in}[x@n])$
    if $\text{Ad}_{\text{out}}[x@n] \neq \text{oldvalue}$ then change = true;
  end
end
Example for Computing Dependences

1 Input (x,y);
2 if (x<0)
3   p=-y;
4 else
5   p=y;
6 z=1
7 while (p!=0)
8   z=z*x
9   p=p-1;
10 Output(z);
Lost of Precision by Directly Computing Aggregate Information Directly

```c
1 x=foo();
2 y=gee();
3 if (…)
4   p=&x;
5   q=&x;
6 else
7   p=&y;
8   q=&y;
9  *p=*q
10  (*(p)());
```

**Distributive analysis:** the aggregation of individual path analysis results is equivalent to computing the aggregate information directly

\[ F(a \land b) = F(a) \land F(b) \]
Summary

- Abstraction domain
- Transfer function
- Termination
- Compute aggregate information directly
  - Precision lost?