Software security, secure programming (and computer forensics)

Lecture 5: Static Analysis (in a nutshell)

Master M2 on Cybersecurity

Academic Year 2016 - 2017

Static Analysis

Main objective:

statically compute some information about (an approximation of) the program behavior

Examples: given (the source-code of) a program P

- does all executions of P satisfy a property φ ?
- does φ satisfied at a given (source) program location ?

$\Rightarrow$ Of course, such questions are undecidable . . . (why ?)

Possible work-arounds:

- over-approximate the pgm behaviour
  $\rightarrow$ result is sound (no false negatives), but incomplete ($\exists$ false positives)
- under-approximate the pgm behaviour
  $\rightarrow$ result is complete (no false negatives), but unsound ($\exists$ false negative)
- non-terminating analysis
  $\rightarrow$ **if** the analysis terminates, **then** the result is sound and complete

What static analysis can be used for ?

General applications

- compiler optimization
  e.g., active variables, available expressions, constant propagations, etc.
- program verification
  e.g., invariant, post-conditions, etc.
- worst-case execution time computation
- parallelization
- etc.

In the “software security” context

- disassembling
  e.g., what are the targets of a dynamic jump
  (`be eax`, content of `eax` ?)
- error and vulnerability detection
  memory error (Null-pointer dereference, out-of-bound array access), use-after-free, arithmetic overflow, etc.

How to proceed ?

Typical problems

- need to reason on a set of executions (not on a single one)

  ex: `x = y * z`

  $\rightarrow$ compute values of `x` for all possible values of `y` and `z` ?
- need to cope with loops

  ex: `while (x < y) do ... end`

  $\rightarrow$ infer the loop behavior for all possible values of `x` and `y` ?

A solution: over-approximate the program behavior

1. propagate an abstract state (over approximating the memory content)

   e.g., $x > 0$, $p \neq NULL$, $x \leq y + z$, p and q are aliases, etc.

   $\rightarrow$ depends on the properties you want to check !
2. **safely** merge memory abstract states produced from $\neq$ paths
3. make loop iterations always finite

Pb: How to find a suitable abstract domains ?

$\rightarrow$ accuracy vs scalability trade-offs . . .

A general framework : abstract interpretation

Although this theory has been invented here in Grenoble . . .

. . . let's jump to Dillig's slides (from UT Austin, Texas) !

Analysis example: Value-Set Analysis

Objective:

compute a (super)-set of possible values of each variable at each program location ...

$Env(x, l)$ = value set of variable `x` at program location `l`

Several possible abstract domains to express these sets:

- bounded value sets (k-sets)
  ex: $Env(x, l) = \{0, 4, 9, 10\}, Env(y, l) = \{1\}, Env(z, l) = \top$
- intervals
  ex: $Env(x, l) = [2, 8], Env(y, l) = [-\infty, 7], Env(z, l) = [-\infty, +\infty]$
- differential bounded matrix (DBM)
  ex : $Env(l) = x - y < 10 \wedge z < 0$
- polyhedra (conjonction of linear equations)
  ex: $Env(l) = x + y \geq 10 \wedge z < 0$
- etc.

VSA with intervals (example 1)

```
1. x := x+y ;
if x>0 then
     2. y:= x + 2
else
     3. y:= -x
4. fi
5. return x+y
```

Asumming (pre-condition) that:

$$x \in [-3, 3], y \in [-1, 5]$$

compute $Env(x, l)$ and $Env(y, l)$ for each program location l
what is the set of return values ?

Computing intervals on expressions

Syntax of expressions

$$e \rightarrow n \mid x \mid e + e \mid e \times e \mid \ldots$$

Computation rules

$Val(e, Env)$ is the interval associated to e within Env

$$\begin{aligned}
Val(n, Env) &= [n, n] \\
Val(x, Env) &= Env(x) \\
Val(e1 + e2, Env) &= [a + c, b + d] \text{ where} \\
&\quad Val(e1, Env) = [a, b] \wedge Val(e2, Env) = [c, d] \\
Val(e1 \times e2, Env) &= [x, y] \text{ where} \\
&\quad Val(e1, Env) = [a, b] \wedge Val(e2, Env) = [c, d] \\
&\quad x = min(a \times c, a \times d, b \times c, b \times d) \\
&\quad y = max(a \times c, a \times d, b \times c, b \times d)
\end{aligned}$$

Intervals propagation

Propagation rules along the statement syntax:

- assignment

$$\{Env1\}\ \texttt{x := e}\ \{Env2\}e$$

where

$$Env2(x) = Val(x, Env1) \wedge Env2(y) = Env1(x) \ \text{ for } \ y \neq x$$

- sequence

$$\{Env1\}\ \texttt{s1; s2}\ \{Env2\}$$

where

$$\{Env1\}\ \texttt{s1}\ \{Env3\} \wedge \{Env3\}\ s2\ \{Env2\}$$

- conditionnal

$$\{Env\}\ \texttt{if (b) then s1 else s2}\ \{Env'\}$$

where

  - {Env $\cap$ Val(cond, Env)} `s1` {Env1}
  - {Env $\cap$ Val($\neg$ cond, Env)} `s2` {Env2}
  - Env' = Env1 $\sqcup$ Env2
    ($Env'(x)$ is the smallest interval containing $Env1(x)$ and $Env2(x)$, $\forall x$)

Iteration ? (example 1)

```
1. x : = 0 ;
while (x < 2) do
  2. x := x+1
3. end
4. return x
```

compute $Env(x, l)$ for each program location l ...

Iteration ? (example 2)

```
1. x : = 0 ;
while (x < 1000) do
  2. x := x+1
3. end
4. return x
```

Compute $Env(x, l)$ for each program location l . . .

What happen if x is initialized to `2000` instead of `1000` ?

Widening

For a lattice $(E, \leq)$, we define $\nabla : E \times E \to E$

∇ is a (pair-)widening operator if and only if

1. Extrapolation:

$$\forall x, y \in E.\ x \leq x \nabla y \wedge y \leq x \nabla y$$

2. Enforce the convergence of $(X_n)_{n \geq 0}$:

$$\begin{aligned} y_0 &= x_0 \\ y_{n+1} &= y_n \nabla x_{n+1} \end{aligned}$$

$(Y_n)_{n \geq 0}$ is ultimately stationnary . . .

$\to$ $\boxed{\text{open “unstable” bounds (jumping over the fix-point) !}}$

Widening on intervals

Definition

$[a, b] \nabla [c, d] = [e, f]$ where,

- e = if $c < a$ then $-\infty$ else a
- f = if $b < d$ then $+\infty$ else b

Examples

- $[2, 3] \nabla [1, 4]$?
- $[1, 4] \nabla [2, 3]$?
- $[1, 3] \nabla [2, 4]$?

Back to the previous example

```
1. x : = 0 ;
while (x < 1000) do
  2. x := x+1
3. end
4. return x
```

$$Env(x,2)_{n+1} = Env(x,2)_n \nabla (Env(x,1)_n \sqcup Env(x,3)_n)$$

$$\begin{array}{rcl} Env(x,2)_1 & = & [0,0] \\ Env(x,2)_2 & = & [0,1] \\ Env(x,2)_3 & = & [0,+\infty] \end{array}$$

$\rightarrow$ stable solution ...but not precise enough ?

$$Env(x,3)_3 = [1000,+\infty]$$

Narrowing

lattice $(E, \leq)$, $\triangle : E \times E \to E$

$\triangle$ is a (pair-)narrowing operator if and only if

1. (abstract) intersection

$$\forall x, y \in E.\ x \cap y \leq x \triangle y$$

2. Enforce the convergence of $(X_n)_{n \geq 0}$:

$$\begin{array}{rcl} y_0 & = & x_0 \\ y_{n+1} & = & y_n \triangle x_{n+1} \end{array}$$

   $(Y_n)_{n \geq 0}$ is ultimately stationnary ...

$\to$ $\boxed{\text{refines open bounds !}}$

Narrowing on intervals

$[a, b] \bigtriangleup [c, d] = [e, f]$ where,

- e = if $a = -\infty$ then c else a
- f = if $b = +\infty$ then d else b

Examples

- $[2, 3] \bigtriangleup [1, +\infty]$?
- $[1, 4] \bigtriangleup [-\infty, 3]$?
- $[1, 3] \bigtriangleup [+\infty, -\infty]$?

Back (again !) to the previous example

```
1. x : = 0 ;
while (x < 1000) do
  2. x := x+1
3. end
4. return x
```

$$Env(x,2)_{n+1} = Env(x,2)_n \bigtriangleup (Env(x,1)_n \cup Env(x,3)_n)$$

$$\begin{array}{rcl} Env(x,2)_1 & = & [0,+\infty] \\ Env(x,2)_2 & = & [0,1000] \end{array}$$

$\rightarrow$ stable solution ...

$$Env(x,3)_2 = [1000,1000]$$

Challenges for static analysis

Accuracy vs scalability trade-off . . .

- inter-procedural analysis (+ recursivity . . .)
- multi-threading
- dynamic memory allocation
- modular reasonning
- libraries (+ legacy code)
- etc.

Application to vulnerability detection ?

Clearly may provide some useful features:

- out-of-bounds array access
- arithmetic overflows
- incorrect memory access (null pointer, mis-aligned address)
- use-after-free
- etc.

But still some limitations:

- exploitability analysis (beyond standard program semantics) ?
- relevant and accurate memory model (for heap and stack)
- self-modifying code (e.g., malwares)
- binary code analysis (see next slide !)

Rk: some useful information on the CERT webpages . . .

Static analysis on binary code

Static analysis relies on a (clear) program semantics

- can be done at the assembly-level (or IR)
- but disassembling is undecidable ...
- ... and disassemblers may rely on static analysis !
  (to retrieve the program CFG)

Static analysis on low-level code is difficult

- no types (a single type for value, addresses, data, code, ...)
- address computation is pervasive ...
  ex: `mov eax, [ecx + 42]`
- function bounds cannot always be retrieved
  $\rightarrow$ many un-initialized memory locations
- sacalability issues
- etc.

Tool examples

Disclaimer: non limitative nor objective list ! (see wikipedia for more info)

Source-level tools

- Astrèe
- Coverity, Polyspace, CodeSonar, HP Fortify, VeraCode
- **Frama-C**, Fluctuat
- etc, etc, . . .

Some binary-level tools

- x86-CodeSurfer
- VeraCode
- Angr
- **BinSec plateform**
- etc ?