Software security, secure programming

Static Analysis (in a nutshell)

Master M2 Cybersecurity

Academic Year 2024 - 2025

Static Analysis

Main objective:

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

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 ?

⇒ Of course, such questions are undecidable ... (why ?)

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 ?
⇒ Of course, such questions are undecidable ... (why ?)

Possible work-arounds:
- ▶ over-approximate the pgm behaviour
  → result is sound (no false negatives), but incomplete ($\exists$ false positives)

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 ?

⇒ Of course, such questions are undecidable ... (why ?)

Possible work-arounds:

- ▶ over-approximate the pgm behaviour
  → result is sound (no false negatives), but incomplete (∃ false positives)
- ▶ under-approximate the pgm behaviour
  → result is complete (no false negatives), but unsound (∃ false negative)

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 ?

⇒ Of course, such questions are undecidable . . . (why ?)

Possible work-arounds:

▶ over-approximate the pgm behaviour
  → result is sound (no false negatives), but incomplete ($\exists$ false positives)

▶ under-approximate the pgm behaviour
  → result is complete (no false negatives), but unsound ($\exists$ false negative)

▶ non-terminating analysis
  → **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.

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.
- ▶ variant analysis (e.g. CodeQL)
  find similar <u>semantic patterns</u> in a code base

Outline

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 ?

How to proceed ?

Typical problems

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

$$ex: x = y \star z$$

$\rightarrow$ compute values of x for all possible values of y and z ?

▶ need to cope with loops

$$ex: \text{while } (x < y) \text{ do } \ldots \text{ end}$$

$\rightarrow$ 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

How to proceed ?

Typical problems

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

ex: $x = y * z$

→ compute values of x for all possible values of y and z ?

► need to cope with loops

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

→ 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.

→ 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 ?

→ accuracy vs scalability trade-offs . . .

Outline

A basic programming language

Syntax

$$
\begin{array}{rcl}
\text{Exp} & ::= & x \mid n \mid \text{op}\,(\text{Exp}, \ldots \text{Exp}) \\
\text{Stm} & ::= & x := \text{Exp} \\
& ::= & \text{Stm ; Stm} \\
& ::= & \text{skip} \\
& ::= & \text{if Exp then Stm else Stm} \\
& ::= & \text{while Exp do Stm end} \\
& ::= & \text{assert Exp}
\end{array}
$$

In practice : arrays, structures, pointers, procedures, etc.

Axiomatic Semantics

$\Rightarrow$ programs viewed as <u>predicate transformers</u> where predicates are <u>assertions</u> on program variables (Hoare, Dijkstra 1976).

▶ Weakest Preconditions (*wp*) : backward computation
   Example :

$$x \geq 0 \ \{x := x + 1;\} \ x > 0$$

▶ Strongest Postcondition (*sp*) : forward computation
   Example :

$$x \geq 0 \ \{x := x + 1;\} \ x > 0$$

Weakest precondition / Strongest postcondition

Let I a statement, P, R, $'$, R' some predicats

The weakest precondition $P = wp(I, R)$ is such that:

$$\forall P' \ (P' \Rightarrow wp(I, R)) \Rightarrow (P' \Rightarrow P)$$

A precondition P' stronger than $x \geq 0 : x > 5$.

Weakest precondition / Strongest postcondition

Let I a statement, P, R, $'$, R' some predicats

The weakest precondition $P = wp(I, R)$ is such that:

$$\forall P' \ (P' \Rightarrow wp(I, R)) \Rightarrow (P' \Rightarrow P)$$

A precondition P' stronger than $x \geq 0 : x > 5$.

The strongest postcondition $R = sp(R, I)$ is such that:

$$\forall R' \ (sp(P, I) \Rightarrow R' \Rightarrow (R \Rightarrow R')$$

A postcondition R' weaker than $x \geq 0 : x > -2$.

Substitution - free/bounded variables

Free and bounded variables

A variable *x* is bounded (resp. free) within formula *F* iff *F* contains an occurrence of *x* which is (resp. which is not) within the scope of a quantifier.

Example:

$$\varphi \equiv P(y, x) \wedge \forall x \, . \, Q(x, y)$$

↪ there is both a free and a bounded occurrence of *x* in φ

Substitution - free/bounded variables

Free and bounded variables

A variable x is bounded (resp. free) within formula F iff F contains an occurrence of x which is (resp. which is not) within the scope of a quantifier.

Example:
$$\varphi \equiv P(y, x) \wedge \forall x . Q(x, y)$$
$\hookrightarrow$ there is both a free and a bounded occurrence of x in φ

Substitution

$P[E/x]$ is the formula P in which all free occurrences of variable x have been replaced by the term E.

Example:
$$(\varphi[x + 1/x])[f/y] \equiv P(f, x + 1) \wedge \forall x . Q(x, f)$$

Computing weakest preconditions: basic instructions

Statement	*def.*	WP
$wp(\text{skip}, R)$	$\triangleq$	R
$wp(x := e, R)$	$\triangleq$	$R[e/x]$
$wp(i_1 \; ; \; i_2, R)$	$\triangleq$	$wp(i_1, wp(i_2, R))$
$wp(\text{assert}(e), R)$	$\triangleq$	$e \wedge R$

Computing weakest preconditions: basic instructions

Statement	*def.*	**WP**
$wp(\text{skip}, R)$	$\hat{=}$	R
$wp(x := e, R)$	$\hat{=}$	$R[e/x]$
$wp(i_1 \; ; \; i_2, R)$	$\hat{=}$	$wp(i_1, wp(i_2, R))$
$wp(\text{assert}(e), R)$	$\hat{=}$	$e \wedge R$

Examples:

1. $wp(x := x + 1, x > 0)$
2. $wp(z := 2 \; ; \; y := z + 1 \; ; \; x := z + y, \; x \in 3..8)$

Another way to write WPs

R
skip;

$R[e/x]$
$\mathbf{x} := \mathbf{e}$;

$wp(i_1, wp(i_2, R))$
$\mathbf{i_1}$;
$wp(i_2, R)$
$\mathbf{i_2}$;

$P \wedge R$
assert(**P**)

Example

$2 + 2 + 1 \in 3..8$
z:=2 ;
$z + z + 1 \in 3..8$
y:=z+1 ;
$z + y \in 3..8$
x:=z+y;
$x \in 3..8$

Computing weakest precondition: conditional statement

$$wp(\text{if } P \text{ then } i_1 \text{else } i_2 \text{ end}, R)$$
$$\hat{=} (P \Rightarrow wp(i_1, R)) \land (\neg P \Rightarrow wp(i_2, R))$$

Computing weakest precondition: conditional statement

$$
\begin{array}{c}
wp(\text{if } P \text{ then } i_1 \text{ else } i_2 \text{ end}, R) \\
\hat{=} (P \Rightarrow wp(i_1, R)) \land (\neg P \Rightarrow wp(i_2, R))
\end{array}
$$

Examples:

- Define $wp(\text{if } e \text{ then } i \text{ end}, R)$.

Computing weakest precondition: conditional statement

$$wp(\text{if } P \text{ then } i_1 \text{else } i_2 \text{ end}, R)$$
$$\hat{=} (P \Rightarrow wp(i_1, R)) \wedge (\neg P \Rightarrow wp(i_2, R))$$

Examples:

▶ Define $wp(\text{if } e \text{ then } i \text{ end}, R)$.

▶ What does the following program compute ? Prove the result ...

```
begin
  if x > y then m := x else m := y end ;
  if z > m then m := z end
end
```

Solution (1)

$(x > y \Rightarrow F_1[x/m]) \land (\neg(x > y) \Rightarrow]F_1[y/m])$ = F_2

if $x > y$

  $F_1[x/m]$

  then $m := x$

  $F_1[y/m]$

  else $m := y$ end ;

 $(z > m \Rightarrow R_1[z/m]) \land (\neg(z > m) \Rightarrow R_1)$ = F_1

if $z > m$

  $R_1[z/m]$;

  then $m := z$

  R_1 ;

  else skip ;

end

 R_1

Solution (2)

Postcondition :

$$(m = x \lor m = y \lor m = z) \land m \geq x \land m \geq y \land m \geq z$$

Let's process $R_1 = m \geq x$.

Computing F_1 :

$$(z > m \Rightarrow m[z/m] \geq x) \land (\neg(z > m) \Rightarrow m \geq x)$$

which can be rewritten:

$$(z > m \Rightarrow z \geq x) \land (\neg(z > m) \Rightarrow m \geq x)$$

Solution (3)

Computing F_2:

$$(x > y \Rightarrow F_1[x/m]) \wedge (\neg(x > y) \Rightarrow F_1[y/m])$$

leading to:

$$
\begin{array}{lll}
(x > y \wedge z > x & \Rightarrow z \geq x) & \wedge \\
(x > y \wedge \neg(z > x) & \Rightarrow x \geq x) & \wedge \\
(\neg(x > y) \wedge z > y & \Rightarrow x \geq x) & \wedge \\
(\neg(x > y) \wedge \neg(z > y) & \Rightarrow y \geq x) &
\end{array}
$$

Each of these 4 propositions is equivalent to **true**.

Computing weakest precondition: iteration

$$wp(\text{while } b \text{ do } S \text{ end}, R) \quad ?$$

Partial correctness

$\rightarrow$ compute the WP **assuming the loop will terminate**

- ▶ need to reason about an arbitrary number of iteration;
- ▶ find a loop invariant I such that:

  1. I is preserved by the loop body:

  $$I \wedge b \Rightarrow wp(S, I)$$

  2. if and when the loop terminates, the post-condition holds:

  $$I \wedge \neg b \Rightarrow R$$

Then

$$wp(\text{while } b \text{ do } S \text{ end}, R) = I$$

Computing weakest precondition: iteration

$$wp(\text{while } b \text{ do } S \text{ end }, R) \quad ?$$

Partial correctness
→ compute the WP **assuming the loop will terminate**

- ▶ need to reason about an arbitrary number of iteration;
- ▶ find a loop invariant I such that:
    1. I is preserved by the loop body:

    $$I \wedge b \Rightarrow wp(S, I)$$

    2. if and when the loop terminates, the post-condition holds:

    $$I \wedge \neg b \Rightarrow R$$

Then

$$wp(\text{while } b \text{ do } S \text{ end }, R) = I$$

Total correctness: prove that the loop **do** terminate . . .
need to introduce a loop variant
(i.e, a measure strictly decreasing at each iteration towards a limit).

Example

Prove the following program using WP

```
{x=n && n>0}
  y := 1 ;
  while x <> 1 do
     y := y*x ;
     x := x-1 ;
   end
{y=n! && n>0}
```

1. WP computation:
   - ▶ based on the program structure (Abstract Syntax Tree)

   - ▶ leaves ⤳ root, following the instruction structure

Implementing WP computation ?

1. WP computation:
   - ▶ based on the program structure (Abstract Syntax Tree)

   - ▶ leaves ⤳ root, following the instruction structure

2. Decidability problems:
   - ▶ simplification and proof of formula
     undecidable in general, heuristics . . .

   - ▶ invariant generation
     undecidable in general, only specific invariant can be generated in some
     restricted conditions (i.e., inductive invariants)

Accurracy vs Effectiveness trade-off

Assertion language

Theories	Complexity	Rappels
First order logic	undecidable	Interactive provers
Booleans	decidable	state enumeration
Intervals	quasi linear	approximation
Polyhedras	exponential	(better) approximation

Tools:
Frama-C/WP (proofs), Frama-C/Value (intervals), Polyspace (polyhedras) . . .

Outline

A general framework : abstract interpretation

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

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) !

Outline

Analysis example: Value-Set Analysis

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

$$Env(x, l) = \text{value set of variable } \texttt{x} \text{ at program location } \texttt{l}$$

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*

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

$$Val(n, Env) = [n, n]$$

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

$$
\begin{aligned}
Val(n, Env) &= [n, n] \\
Val(x, Env) &= Env(x)
\end{aligned}
$$

Computing intervals on expressions

Syntax of expressions

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

Computation rules

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

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

Computing intervals on expressions

Syntax of expressions

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

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

$$\{Env1\} \; \mathrm{x} := \mathrm{e} \; \{Env2\}$$

where

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

Intervals propagation

Propagation rules along the statement syntax:

- **assignment**

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

  where

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

- **sequence**

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

  where

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

Intervals propagation

Propagation rules along the statement syntax:

▶ assignment
$$\{Env1\} \; x := e \; \{Env2\}$$

where

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

▶ sequence
$$\{Env1\} \; s1; s2 \; \{Env2\}$$

where

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

▶ conditionnal
$$\{Env\} \; \text{if } (b) \text{ then } s1 \text{ else } s2 \; \{Env'\}$$

where
- ▶ {Env ∩ Val(b, Env)} s1 {Env1}
- ▶ {Env ∩ Val(¬ b, Env)} s2 {Env2}
- ▶ Env' = Env1 ⊔ 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, where . . .

$$Env(x, 2) = Env(x, 1) \sqcup Env(x, 3)$$

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, where ...

$$Env(x, 2) = Env(x, 1) \sqcup Env(x, 3)$$

Actually, what we aim to compute is the least solution of function Env, i.e:

$$Env^0(\bot, l) \ \sqcup \ Env^1(\bot, l) \ \sqcup \ Env^2(\bot, l) \ \sqcup \ \ldots \ \sqcup Env^k(\bot, l) \ \sqcup \ \ldots$$

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 ...

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 happens if we replace `x := x+1` by `x := x-1` ?

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 happens if we replace x := x+1 by x := x−1 ?

How to cope with such **loooong**, or even **infinite**, computations ?

A practical solution : Widening & Narrowing operators

See Hakjoo Oh slides . . .

Outline

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.

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 !)

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 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.

What help for "security analysis" ?

"security analysis" = vulnerability detection

A pragmatic approach:

1. annotate the code with "vulnerability checks" (e.g., `frama-c -rte`)
   i.e., assertions to detect integer overflows, invalid memory accesses
   (arrays, pointers), etc

What help for "security analysis" ?

"security analysis" = vulnerability detection

A pragmatic approach:

1. annotate the code with "vulnerability checks" (e.g., `frama-c -rte`)
   i.e., assertions to detect integer overflows, invalid memory accesses
   (arrays, pointers), etc

2. run a VSA
   → reveals a lot of hot spots (= unchecked assertions)

What help for "security analysis" ?

"security analysis" = vulnerability detection

A pragmatic approach:

1. annotate the code with "vulnerability checks" (e.g., `frama-c -rte`)
   i.e., assertions to detect integer overflows, invalid memory accesses
   (arrays, pointers), etc

2. run a VSA
   → reveals a lot of hot spots (= unchecked assertions)

3. add user-defined assertions when possible . . .
   e.g., function pre/post conditions, loop invariants, extra information . . .
   → consider **proving** (some of) these assertions ?

What help for "security analysis" ?

"security analysis" = vulnerability detection

A pragmatic approach:

1. annotate the code with "vulnerability checks" (e.g., `frama-c -rte`)
   i.e., assertions to detect integer overflows, invalid memory accesses
   (arrays, pointers), etc

2. run a VSA
   $\rightarrow$ reveals a lot of hot spots (= unchecked assertions)

3. add user-defined assertions when possible . . .
   e.g., function pre/post conditions, loop invariants, extra information . . .
   $\rightarrow$ consider **proving** (some of) these assertions ?

4. run the VSA again . . .

What help for "security analysis" ?

"security analysis" = vulnerability detection

A pragmatic approach:

1. annotate the code with "vulnerability checks" (e.g., `frama-c -rte`)
   i.e., assertions to detect integer overflows, invalid memory accesses
   (arrays, pointers), etc

2. run a VSA
   $\rightarrow$ reveals a lot of hot spots (= unchecked assertions)

3. add user-defined assertions when possible . . .
   e.g., function pre/post conditions, loop invariants, extra information . . .
   $\rightarrow$ consider **proving** (some of) these assertions ?

4. run the VSA again . . .

$\Rightarrow$ a set of potential vulnerabilities remains, to be discharged by other means,
possibly on a **program slice**
(false positive ? real bug but harmless w.r.t security ? real vulnerability ?)

What help for "security analysis" ?

"security analysis" = vulnerability detection

A pragmatic approach:

1. annotate the code with "vulnerability checks" (e.g., `frama-c -rte`)
   i.e., assertions to detect integer overflows, invalid memory accesses
   (arrays, pointers), etc

2. run a VSA
   $\rightarrow$ reveals a lot of hot spots (= unchecked assertions)

3. add user-defined assertions when possible ...
   e.g., function pre/post conditions, loop invariants, extra information ...
   $\rightarrow$ consider **proving** (some of) these assertions ?

4. run the VSA again ...

$\Rightarrow$ a set of potential vulnerabilities remains, to be discharged by other means,
possibly on a **program slice**
(false positive ? real bug but harmless w.r.t security ? real vulnerability ?)

Rk: some static analysis tools also provide bug finding facilities
(i.e., no false postives, ... but false negatives instead)

Tool examples

Source-level tools

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

You can see also:

- ▶ the CERT webpages
- ▶ the Microsoft "Secure Development Lifecycle" . . .