Software security, secure programming (and computer forensics)

Lecture 9: from Static Analysis to (Dynamic) Symbolic Execution

Master M2 on Cybersecurity

Academic Year 2016 - 2017

Summary

Static analysis techniques

- allow to (automatically) reason about a whole program without executing it ...
- but at the price of approximations due to undecidability problems:
  - over-approximations $\rightsquigarrow$ false positives
  - under-approximations $\rightsquigarrow$ false negatives
- example: **value-set analysis (VSA)**
  abstract representation = trade-off between accuracy and efficiency
  (e.g., intervals vs polyhedra vs ...)
- can be leveraged with use-provided asertions ...
  (to deal with library calls, “complex” code patterns, etc.)

Summary

Static analysis techniques

- allow to (automatically) reason about a whole program without executing it . . .
- but at the price of approximations due to undecidability problems:
  - over-approximations $\rightsquigarrow$ false positives
  - under-approximations $\rightsquigarrow$ false negatives
- example: **value-set analysis (VSA)**
  abstract representation = trade-off between accuracy and efficiency
  (e.g., intervals vs polyhedra vs . . .)
- can be leveraged with use-provided asertions . . .
  (to deal with library calls, “complex” code patterns, etc.)

Long (success) story in program verification $\Rightarrow$ numerous tools available!

Summary

Static analysis techniques

- allow to (automatically) reason about a whole program without executing it . . .
- but at the price of approximations due to undecidability problems:
  - over-approximations $\rightsquigarrow$ false positives
  - under-approximations $\rightsquigarrow$ false negatives
- example: **value-set analysis (VSA)**
  abstract representation = trade-off between accuracy and efficiency
  (e.g., intervals vs polyhedra vs . . .)
- can be leveraged with use-provided asertions . . .
  (to deal with library calls, “complex” code patterns, etc.)

Long (success) story in program verification $\Rightarrow$ numerous tools available!

But:

- not so effective on binary code, simple memory model
- not go “beyond the bug” ($\neq$ exploitability analysis)
- may provide too many false postives ?

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
   $\rightarrow$ 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
   $\rightarrow$ 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 ?

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
   → 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 ?
4. run the VSA again . . .

⇒ 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)

Today's menu

1. A few words on assertion proving using weakest pre-conditions (WP)

2. Some exercices on VSA (and WP)

3. An alternative/complementary approach to static analysis:

   (Dynamic) Symbolic Execution

   - may help to discharge/confirm unchecked assertions
   - may help to detect (others) vulnerabilities . . .
     (in a more general context)

A basic programming language

Syntax

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

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

Axiomatic Semantics

$\Rightarrow$ programs viewed as *predicate transformers* where predicates are *assertions* 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:

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

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

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($\mathbf{P}$)

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

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

Computing weakest precondition: conditional statement

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

Examples:

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

Computing weakest precondition: conditional statement

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

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)

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

if $x > y$

$F_1[x/m]$

then $m := x$

$F_1[y/m]$

else $m := y$ end ;

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

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:

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

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

$\rightarrow$ 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

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

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}
```

Implementing WP computation ?

1. WP computation:
   - based on the program structure (Abstract Syntax Tree)
   - leaves $\rightsquigarrow$ root, following the instruction structure

Implementing WP computation ?

1. WP computation:
   - based on the program structure (Abstract Syntax Tree)
   - leaves $\rightsquigarrow$ 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) . . .

Static analysis . . . what else ?

Another (quite) standard technique for program validation: run tests . . . !

But, not always easy to find “good” test inputs ?

Example: which input allow to activate the vulnerability below ?

```
int twice(int v) {
  return 2 * v;
}

void test(int x, int y) {
  int *t = (int *) malloc((x+10) * sizeof(int)) ;
  z = twice(y);
  if (x == z) {
       assert (y <= x +10) ;
       assert (y > 0) ;
   t[y] = 0 ;
  }
}
```

Static analysis . . . what else ?

Another (quite) standard technique for program validation: run tests . . . !

But, not always easy to find "good" test inputs ?

Example: which input allow to activate the vulnerability below ?

```
int twice(int v) {
  return 2 * v;
}

void test(int x, int y) {
  int *t = (int *) malloc((x+10) * sizeof(int)) ;
  z = twice(y);
  if (x == z) {
       assert (y <= x +10) ;
       assert (y > 0) ;
   t[y] = 0 ;
  }
}
```

A random search may not succeed . . .
Can *"static analysis like techniques"* help ?
⇒ An (old !) answer: symbolic execution . . .

Symbolic Excecution

King, 76

Objective:

run a program paths (as in test execution) but mapping variables to symbolic values (instead of **concrete ones**)

- each symbolic execution allows to reason on a set of concrete executions
  (all the ones following **the same path** in the CFG)
- allow to decide if a CFG path is feasable or not (and with wich input values)
- allow to explore a **(finite !)** set of paths in the CFG . . .

Symbolic Excecution

King, 76

Objective:

run a program paths (as in test execution) but mapping variables to symbolic values (instead of **concrete ones**)

- each symbolic execution allows to reason on a set of concrete executions
  (all the ones following **the same path** in the CFG)
- allow to decide if a CFG path is feasable or not (and with wich input values)
- allow to explore a **(finite !)** set of paths in the CFG . . .

Principle:

Associate a path predicate φ_σ to each path σ of the CFG:

$$(\exists \text{ a variable valuation } v \text{ s.t } v \models \varphi_\sigma) \Leftrightarrow (v \text{ covers } \sigma)$$

(φ_σ is the conjunction of all boolean conditions associated to σ in the CFG)

- solving φ_σ indicates if σ is feasible
- iteration over a finite subset of the CFG paths . . .

In practice: express φ_σ in a decidable logic fragment (e.g., SMT).

More on Symbolic Execution . . .

- application to the previous example
- what can we do if:
  - the **path predicate** cannot be expressed in a decidable logic ? (e.g., non linear operations)
  - the program contains conditions on non-reversible functions ? (e.g., `if (x == hash(y)) ...`)
  - part of the program code is not available (e.g., library functions, `if (!strcmp(s1, s2) ...`)

$\rightarrow$ combine symbolic and concrete executions:
concolic execution (or Dynamic Symbolic Execution)

see that on Martin Vechev's slides . . .

Conclusion about Dynamic Symbolic Execution

- an effective test generation and test execution technique
  - can be used on "arbitrary" code
    dynamic allocation, complex math. functions, binary code
  - trade-off between correctness, completeness and efficiency
    (ratio between symbolic and concrete values)
  - can be used in a coverage-oriented (bug finding) or path-oriented
    (vulnerability confirmation) way

$\boxed{\Rightarrow \text{widely used in security \ldots (and also for exploitability analysis)}}$

Conclusion about Dynamic Symbolic Execution

- an effective test generation and test execution technique
  - can be used on “arbitrary” code
    dynamic allocation, complex math. functions, binary code
  - trade-off between correctness, completeness and efficiency
    (ratio between symbolic and concrete values)
  - can be used in a coverage-oriented (bug finding) or path-oriented
    (vulnerability confirmation) way

$\boxed{\Rightarrow \text{widely used in security \ldots (and also for exploitability analysis)}}$

- numerous existing tools ...

Conclusion about Dynamic Symbolic Execution

- an effective test generation and test execution technique
  - can be used on "arbitrary" code
    dynamic allocation, complex math. functions, binary code
  - trade-off between correctness, completeness and efficiency
    (ratio between symbolic and concrete values)
  - can be used in a coverage-oriented (bug finding) or path-oriented
    (vulnerability confirmation) way

$\boxed{\Rightarrow \text{widely used in security \dots (and also for exploitability analysis)}}$

- numerous existing tools ...
- however, not all problems solved (yet ?), e.g.:
  - "path explosion" problem
  - can be rather slow (compared with *fuzzing*)