UFR IM²AG

UNIVERSITÉ Grenoble Alpes

Grenoble INP Ensimag

Software security, secure programming

A brief introduction to Frama-C

Master M2 Cybersecurity

Academic Year 2024 - 2025

The Frama-C plateform

An open-source collaborative plateform for the analysis of C programs
`http://frama-c.com/index.html`

- developed by the CEA List and INRIA Saclay
- offers an integrated set of code analysis plug-ins:
  - runtime-error detection (RTE)
    $\hookrightarrow$ annotate the code with assertions ensuring absence of runtime errors and undefined behaviors;
  - value analysis (EVA)
    $\hookrightarrow$ over-approximate of the program behavior
  - weakest-precondition computations (WP)
    $\hookrightarrow$ semi-automated proof technique of program properties
  - dependency analysis and slicing
  - control-flow-grah and call-graph computations
  - etc.

$\rightarrow$ we are going to use essentially RTE, EVA, and (possibly) WP . . .

Value-Analysis[1]

Goal: staticaly compute **an over-approximated** set of values, for each variable, at each program location.

Principle

Abstract Interpretation

- analyze the program behavior using an **abstract semantics** (i.e., based on an abstract domains to express values and operations)
- loop behaviors are over-approximated as fix-point computation, termination being accelerated/enforced using widening & narrowing operators.

Outcomes

- help to detect potential runtime errors (arithmetic overflow, invalid memory access, etc.)
- may produce **false positives** (i.e., non existing bugs) when the over-approximation is too coarse . . .

[1] (more to come during the next lecture !)

WP computations[2]

Principle

Weakest-Precondition computations

- Given a program P and a property Ψ, "compute" the more general pre-condition Φ on P "inputs" such that the (post-condition) Ψ holds if/when P terminates;
- Not a fully automated computation, **loop invariants** and **loop termination** arguments may have to be user-provided ...

Outcomes

- help to **refine** the results provided by EVA, adding more precise information on the program behavior;
- still limited by the user-provided information and the underlying solver capabilites ...

[2] (more to come during the next lecture !)

Using Frama-C in our "software security" context: possible workflow

1. Generate the runtime assertions (Rtegen)

   ```
   frama-c-gui -rte example.c
   ```

   $\rightarrow$ verify that you understand them . . .

Using Frama-C in our "software security" context: possible workflow

1. Generate the runtime assertions (Rtegen)

   ```
   frama-c-gui -rte example.c
   ```

   → verify that you understand them . . .

2. Run the value analysis (EVA)

   ```
   frama-c-gui -eva example.c
   ```

   → verify that you understand the results
   Why some (obvious ?) assertions may not be validated ?

Using Frama-C in our “software security” context: possible workflow

1. Generate the runtime assertions (Rtegen)

   ```
   frama-c-gui -rte example.c
   ```

   → verify that you understand them . . .

2. Run the value analysis (EVA)

   ```
   frama-c-gui -eva example.c
   ```

   → verify that you understand the results
   Why some (obvious ?) assertions may not be validated ?

3. If you thing the code is incorrect/unsecure, try to strengthen it and goto 1

Using Frama-C in our "software security" context: possible workflow

1. Generate the runtime assertions (Rtegen)

   ```
   frama-c-gui -rte example.c
   ```

   → verify that you understand them ...

2. Run the value analysis (EVA)

   ```
   frama-c-gui -eva example.c
   ```

   → verify that you understand the results
   Why some (obvious ?) assertions may not be validated ?

3. If you thing the code is incorrect/unsecure, try to strengthen it and goto 1

4. Otherwise, if you think the code is correct:
   - try to add some extra assertions (and loop invariants ?)
   - optionally, try to use `WP` to prove them ?
   - re-run EVA with these new assertions ...

All these plugins can also be conveniently accessed through the `Analyses` menu (`Rtegen`, `Eva` and `WP`) of the graphical user interface:

```
frama-c-gui example.c
```

More on the value analysis plug-in

(Evolved) Value Analysis

- Based on Abstract Interprattion to compute abstract variable domains
- Fully automated, but can be user-guided through ACSL annotations
- mainly used to discharge runtime-error asssertions (RTE), but internaly used by other plugins . . .

Some practical informations

- abstract domains = value sets and intervals (non relational domains)
- controlling approximations (*time vs memory*)
  - syntactic loop unrolling (`-ulevel`)
  - semantic unrolling (`-slevel`)
    $\rightarrow$ useful when widenning operators are too coarse
  - adding ACSL loop invariants, or extra assertions . . .

More on WP: expressing assertions with ACSL

Ansi-C Specification Language

- first order logic
- use C types (int, float, pointers, arrays, etc.) + Z + R
- built-in predicates for memory access: `valid, separated`
  $\rightarrow$ allows to express memory-level requirements (beyond the C semantics)
- used as special comments:

  ```
  /*@ .......... */
  ```

$\Rightarrow$ have a look to the short tutorial:

`http://frama-c.com/acsl_tutorial_index.html`

Example of assertion

- valid memory access:
  `\valid(a)` means that address `a` refers to
  a memory location **correctly allocated** (w.r.t. the C type of `a`)

```
      \valid(p)
      \valid(t+i)
 \valid(t+)(0..n-1)
```

- pre- and post- conditions

```
\requires x<= n && \valid(t+x)
\ensures (t+x) = x
```

- loop invariants, assertions

```
loop invariant z==x+y
assert x>=0
```

- etc.

Lab Session

Objective:

Evaluate the strengths and weaknesses of static analysis tools (like Frama-C) for source-level vulnerability detection . . .

1. Play with the examples/exercices provided in the course web page . . .
2. You can also use Frama-C on the “grub” example (in addition with AFL++, Klee, etc.)