# YAML-Based Format for Violation Witnesses

(work in progress)

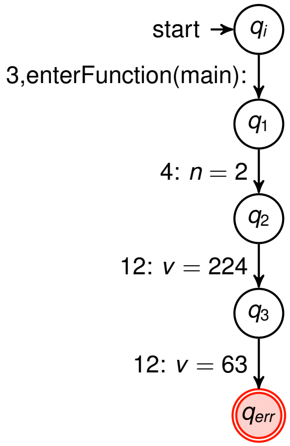
Paulína Ayaziová, Dirk Beyer, Marian Lingsch Rosenfeld, Martin Spießl, and Jan Strejček

VeWit 2023

July 17, 2023

## Existing format

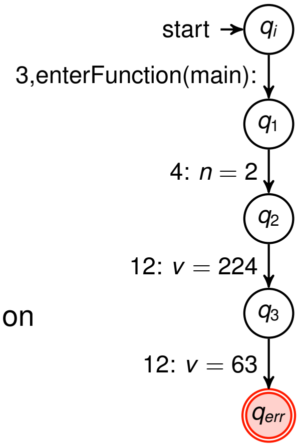
- introduced in 2015
- based on GraphML


$$\begin{tikzcd}[row sep=large]
\text{start} \arrow[r] & q_i \arrow[d, "\text{3,enterFunction(main):}"'] \\
& q_1 \arrow[d, "4{:}\ n = 2"'] \\
& q_2 \arrow[d, "12{:}\ v = 224"'] \\
& q_3 \arrow[d, "12{:}\ v = 63"'] \\
& q_{err}
\end{tikzcd}$$

## Existing format

- introduced in 2015
- based on GraphML

+ widely accepted by the community
+ improved the quality of verification tools
+ other applications, e.g. cooperative verification

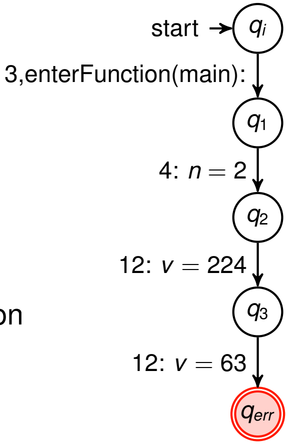

$$\begin{tikzcd}
\text{start} \arrow[r] & q_i \arrow[d, "\text{3,enterFunction(main):}"'] \\
& q_1 \arrow[d, "4:\ n = 2"'] \\
& q_2 \arrow[d, "12:\ v = 224"'] \\
& q_3 \arrow[d, "12:\ v = 63"'] \\
& q_{err}
\end{tikzcd}$$

## Existing format

- introduced in 2015
- based on GraphML

+ widely accepted by the community
+ improved the quality of verification tools
+ other applications, e.g. cooperative verification

– witness validators do not support all features
– verifiers do not use the whole power of the format
– semantics given on Control Flow Automata (CFA), but translation to CFA is ambiguous



## Our goals

- design a format for violation witnesses with a clear semantics on source code
- develop validators that fully implement the format

## Our goals

- design a format for violation witnesses with a clear semantics on source code
- develop validators that fully implement the format

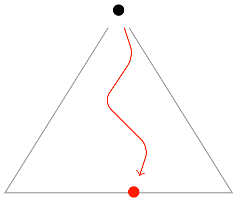
$\Longrightarrow$ the format needs to be as simple as possible

## Design decisions

- start with the support of the most common properties and sequential programs and then extend it
- integrate the format to the existing YAML format for correctness witnesses

## Design decisions

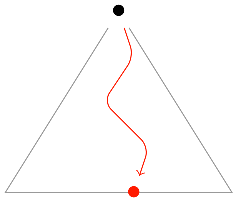
How many runs should a violation witness describe?



single violating run

## Design decisions

How many runs should a violation witness describe?

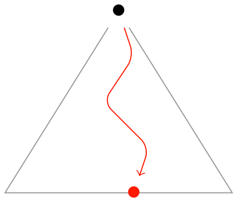


single violating run

- a lot of detailed information
  - values of all inputs
  - order of evaluation: `f(x) + g(y)`
  - addresses of allocations: `p = malloc(10)`
  - ...

## Design decisions

How many runs should a violation witness describe?

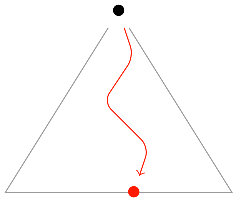


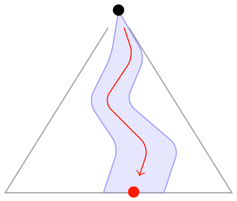
single violating run

- a lot of detailed information
  - values of all inputs
  - order of evaluation: `f(x) + g(y)`
  - addresses of allocations: `p = malloc(10)`
  - ...
- big and hard to validate

## Design decisions

How many runs should a violation witness describe?


single violating run


multiple runs,
at least one violating ✓

- a lot of detailed information
  - values of all inputs
  - order of evaluation: `f(x) + g(y)`
  - addresses of allocations: `p = malloc(10)`
  - ...
- big and hard to validate

## Waypoints

waypoint = basic element of witnesses

## Waypoints

waypoint = basic element of witnesses

each waypoint has 4 aspects:

- action - the meaning in the witness
- location - code location it is associated to
  - filename
  - line number
  - column (optional, the default is the first suitable column)
- type - the type of constraint it puts on runs
- constraint - the constraint itself

## Waypoint types

1. assumption
   - location: before or after a statement
   - constraint: a side-effect-free expression
   - for example `x[5] > z + 5` or `ptr != NULL`

## Waypoint types

1. assumption
   - location: before or after a statement
   - constraint: a side-effect-free expression
   - for example `x[5] > z + 5` or `ptr != NULL`
2. branching
   - location: branching keyword like `if`, `while`, . . .
   - constraint: `true` or `false`

## Waypoint types

1. assumption
   - location: before or after a statement
   - constraint: a side-effect-free expression
   - for example `x[5] > z + 5` or `ptr != NULL`
2. branching
   - location: branching keyword like `if`, `while`, ...
   - constraint: `true` or `false`
3. identifier_evaluation
   - location: an identifier (currently only function call)
   - constraint: `true` (default value)

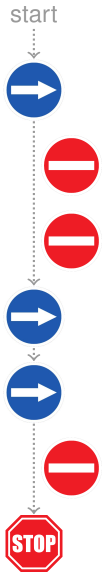
## Waypoint types

1. assumption
   - location: before or after a statement
   - constraint: a side-effect-free expression
   - for example `x[5] > z + 5` or `ptr != NULL`
2. branching
   - location: branching keyword like `if`, `while`, ...
   - constraint: `true` or `false`
3. identifier_evaluation
   - location: an identifier (currently only function call)
   - constraint: `true` (default value)
4. function_return
   - location: the right parenthesis after the function call
   - constraint: `\return` *op* *const*, where $op \in \{==, !=, <=, \ldots\}$ and *const* is a constant

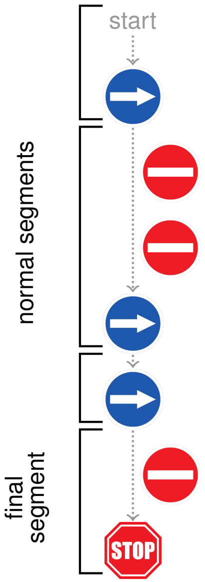
## Waypoint actions

- follow - the waypoint has to be passed as soon as the location is entered
- avoid - the run represented by the witness must not pass the waypoint (“sink node”)
- target - the property violation

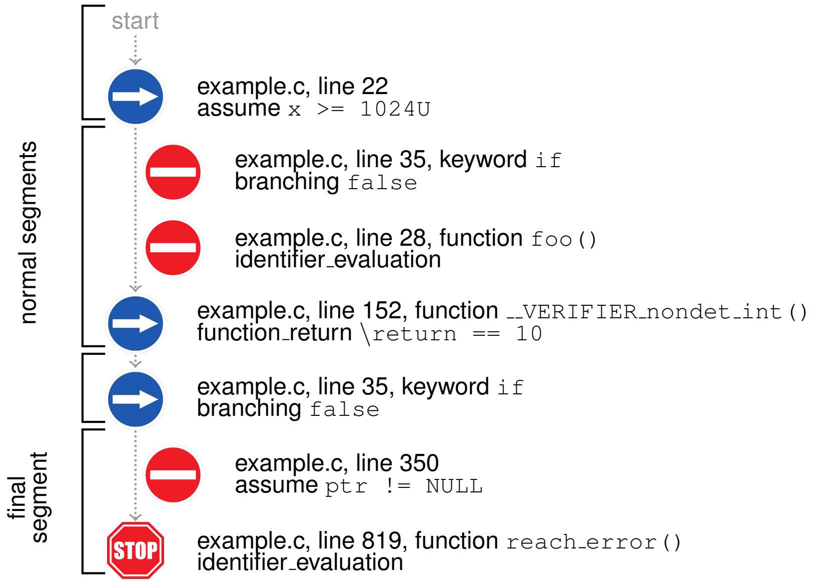
## Witness example



## Witness example



## Witness example



example.c, line 22
assume `x >= 1024U`

example.c, line 35, keyword `if`
branching `false`

example.c, line 28, function `foo()`
identifier_evaluation

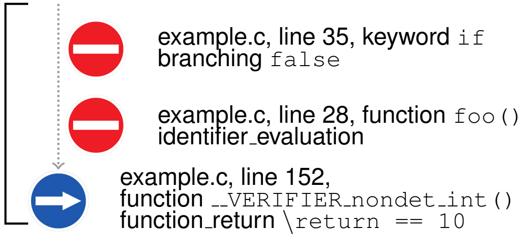
example.c, line 152, function `__VERIFIER_nondet_int()`
function_return `\return == 10`

example.c, line 35, keyword `if`
branching `false`

example.c, line 350
assume `ptr != NULL`

example.c, line 819, function `reach_error()`
identifier_evaluation

# Witness example - YAML notation


example.c, line 35, keyword if
branching false
example.c, line 28, function foo()
identifier_evaluation
example.c, line 152,
function __VERIFIER_nondet_int()
function_return \return == 10

```
- segment:
  - waypoint:
    action: avoid
    type: branching
    location:
      file_name: example.c
      line: 35
    constraint:
      string: false
  - waypoint:
    action: avoid
    type: identifier_evaluation
    location:
      file_name: example.c
      line: 28
  - waypoint:
    action: follow
    type: function_return
    location:
      file_name: example.c
      line: 152
    constraint:
      string: \return == 10
```

## Current status and future plans

- informal textual description and an example of a witness
- supports only violation witnesses of reachability safety, memory safety, and no overflows on sequential programs
- three independent validators under development
- support of this format by several verifiers under development

## Current status and future plans

- informal textual description and an example of a witness
- supports only violation witnesses of reachability safety, memory safety, and no overflows on sequential programs
- three independent validators under development
- support of this format by several verifiers under development

**plans**

- write a documentation (expected availability in September 2023)
- use it in SV-COMP 2023 (in addition to the GraphML format)
- extend it to support parallel programs and other properties
- add some features that will be requested by the community (repetitions of segment, Kleene star, . . . ?)

## Conclusion

- new violation witness format in YAML
- very simple and easy to implement
- easier to read by humans
- clear semantics
- extendable