Specification of Concretization and Symbolization Policies in Symbolic Execution

Sébastien Bardin
joint work with
Robin David, Josselin Feist, Laurent Mounier, Marie-Laure Potet, Thanh Dihn Ta, Jean-Yves Marion

CEA LIST (Paris-Saclay, France)

ISSTA 2016

Takeaway

Dynamic Symbolic Execution (DSE) : powerful approach to verif. and testing

- three key ingredients : path predicate computation & solving, path search, concretization & symbolization policy (C/S)

C/S is an essential part, yet mostly not studied

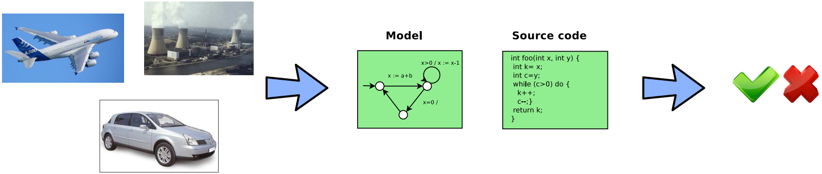
- many policies (one per tool), no systematic study of C/S
- undocumented, unclear
- tools : often a single hardcoded policy, no reuse across tools

Our goal : establish C/S as a proper field of study [focus first on specification]

- CSML, a specification language for C/S ✓
  - clear, non-ambiguous [documentation]
  - tool independent [reuse, sharing, tuning]
  - executable [input for tools]
- implemented in BINSEC ✓
- an experimental comparison of C/S policies ✓

About formal verification

- Between Software Engineering and Theoretical Computer Science
- Goal = proves correctness in a mathematical way



Key concepts : $M \models \varphi$

- M : semantic of the program
- φ : property to be checked
- $\models$: algorithmic check

Kind of properties

- absence of runtime error
- pre/post-conditions
- temporal properties

From (a logician's) dream to reality

Industrial reality in some key areas, especially safety-critical domains

- hardware, aeronautics [airbus], railroad [metro 14], smartcards, drivers [Windows], certified compilers [CompCert] and OS [Sel4], etc.

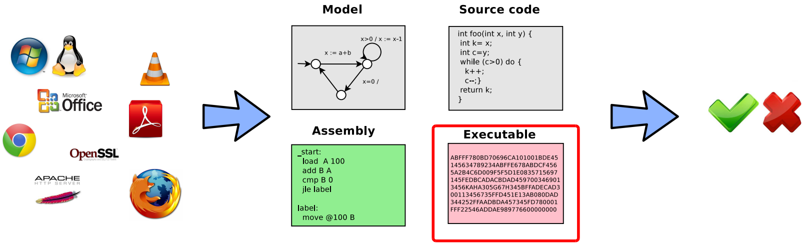
Ex : Airbus

Verification of

- runtime errors [Astrée]
- functional correctness [Frama-C]
- numerical precision [Fluctuat]
- source-binary conformance [CompCert]
- ressource usage [Absint]

Next big challenge

- Apply formal methods to less-critical software
- Very different context : no formal spec, less developer involvement, etc.

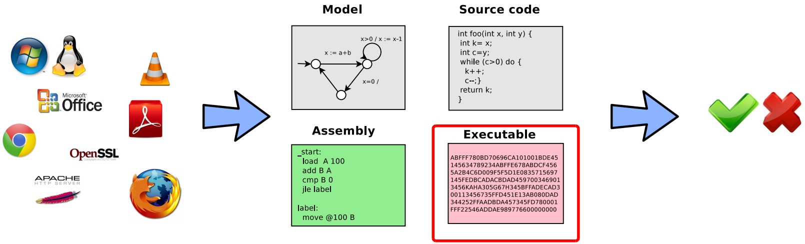


Difficulties

- robustness [w.r.t. software constructs]
- no place for false alarms
- scale
- sometimes, not even source code

Next big challenge

- Apply formal methods to less-critical software
- Very different context : no formal spec, less developer involvement, etc.



Difficulties

- robustness [w.r.t. software constructs]
- no place for false alarms
- scale
- sometimes, not even source code

DSE as a first step

- very robust
- (mostly) no false alarm
- scale in some ways
- ok for binary code

Introducing DSE

Dynamic Symbolic Execution [since 2004-2005 : dart, cute, pathcrawler]

- a very powerful formal approach to verification and testing
- many tools and successful case-studies since mid 2000's
  - SAGE, Klee, Mayhem, etc.
  - coverage-oriented testing, bug finding, exploit generation, reverse
- arguably one of the most wide-spread use of formal methods

Very good properties

- mostly no false alarm, robust, scale, ok for binary code

Introducing DSE

Dynamic Symbolic Execution [since 2004-2005 : dart, cute, pathcrawler]

- a very powerful formal approach to verification and testing
- many tools and successful case-studies since mid 2000's
  - SAGE, Klee, Mayhem, etc.
  - coverage-oriented testing, bug finding, exploit generation, reverse
- arguably one of the most wide-spread use of formal methods

Very good properties

- mostly no false alarm, robust, scale, ok for binary code

Key idea : path predicate [King 70's]

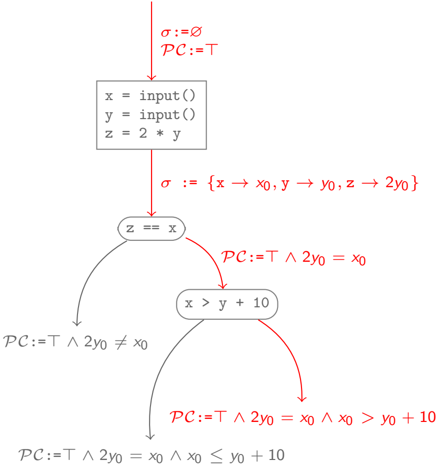
- consider a program P on input v, and a given path σ
- a path predicate φ_σ for σ is a formula s.t.

  $\mathtt{v} \models \varphi_\sigma \Rightarrow \mathtt{P(v)}$ follows σ
- intuitively the **conjunction of all branching conditions**
- old idea, recent renew interest [powerful solvers, dynamic+symbolic]

DSE

```
int main () {
    int x = input();
    int y = input();
    int z = 2 * y;
    if (z == x) {
        if (x > y + 10)
            failure;
    }
    success;
}
```

- given a path of the program
- automatically find input that follows the path
- then, iterate over all paths

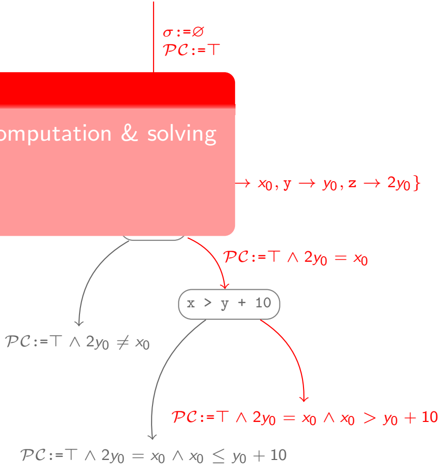


DSE

```
int main () {
    int x = input();
    int y = input(
    int z = 2 * y;
    if (z == x) {
        if (x > y
            failur
    }
    success;
}
```

Three key ingredients

- path predicate computation & solving
- path search
- C/S policy



- given a path of the program
- automatically find input that follows the path
- then, iterate over all paths

Path predicate computation

Usually easy to compute [forward, introduce new logical variables at each step]

Loc	Instruction
0	input(y,z)
1	w := y+1
2	x := w + 3
3	if (x < 2 * z) [True branch]
4	if (x < z) [False branch]

Path predicate (input Y_0 et Z_0)

Path predicate computation

Usually easy to compute [forward, introduce new logical variables at each step]

Loc	Instruction
0	input(y,z)
1	w := y+1
2	x := w + 3
3	if (x < 2 * z) [True branch]
4	if (x < z) [False branch]

Path predicate (input Y_0 et Z_0)
let $W_1 \triangleq Y_0 + 1$ in

Path predicate computation

Usually easy to compute [forward, introduce new logical variables at each step]

Loc	Instruction
0	input(y,z)
1	w := y+1
2	x := w + 3
3	if (x < 2 * z) [True branch]
4	if (x < z) [False branch]

Path predicate (input Y_0 et Z_0)

let $W_1 \triangleq Y_0 + 1$ in

let $X_2 \triangleq W_1 + 3$ in

Path predicate computation

Usually easy to compute [forward, introduce new logical variables at each step]

Loc	Instruction
0	input(y,z)
1	w := y+1
2	x := w + 3
3	if (x < 2 * z) [True branch]
4	if (x < z) [False branch]

Path predicate (input Y_0 et Z_0)

let $W_1 \triangleq Y_0 + 1$ in
  let $X_2 \triangleq W_1 + 3$ in
    $X_2 < 2 \times Z_0$

Path predicate computation

Usually easy to compute [forward, introduce new logical variables at each step]

Loc	Instruction
0	input(y,z)
1	w := y+1
2	x := w + 3
3	if (x < 2 * z) [True branch]
4	if (x < z) [False branch]

Path predicate (input Y_0 et Z_0)

let $W_1 \triangleq Y_0 + 1$ in

let $X_2 \triangleq W_1 + 3$ in

$X_2 < 2 \times Z_0 \wedge X_2 \geq Z_0$

Path Exploration

input : a program P

output : a test suite TS covering all feasible paths of $Paths^{\leq k}(\mathtt{P})$

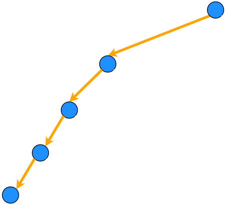
- pick a path $\sigma \in Paths^{\leq k}(\mathtt{P})$
- compute a *path predicate* φ_σ of σ
- solve φ_σ for satisfiability
- SAT(s) ? get a new pair $<$ s, σ $>$
- loop until no more path to cover

Path Exploration

input : a program P

output : a test suite *TS* covering all feasible paths of $Paths^{\leq k}(\texttt{P})$

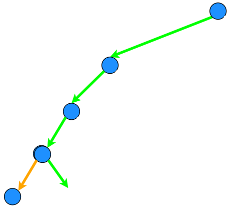
- pick a path $\sigma \in Paths^{\leq k}(\texttt{P})$
- compute a *path predicate* φ_σ of σ
- solve φ_σ for satisfiability
- SAT(s) ? get a new pair $< s, \sigma >$
- loop until no more path to cover



Path Exploration

input : a program P
output : a test suite *TS* covering all feasible paths of $Paths^{\leq k}$(P)

- pick a path $\sigma \in Paths^{\leq k}$(P)
- compute a *path predicate* φ_σ of σ
- solve φ_σ for satisfiability
- SAT(s) ? get a new pair $< s, \sigma >$
- loop until no more path to cover

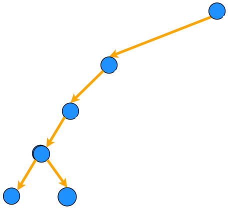


Path Exploration

input : a program P

output : a test suite *TS* covering all feasible paths of $Paths^{\leq k}(\texttt{P})$

- pick a path $\sigma \in Paths^{\leq k}(\texttt{P})$
- compute a *path predicate* φ_σ of σ
- solve φ_σ for satisfiability
- SAT(s) ? get a new pair $< s, \sigma >$
- loop until no more path to cover

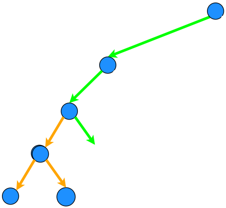


Path Exploration

input : a program P

output : a test suite TS covering all feasible paths of $Paths^{\leq k}(\mathtt{P})$

- pick a path $\sigma \in Paths^{\leq k}(\mathtt{P})$
- compute a *path predicate* φ_σ of σ
- solve φ_σ for satisfiability
- SAT(s) ? get a new pair $< s, \sigma >$
- loop until no more path to cover

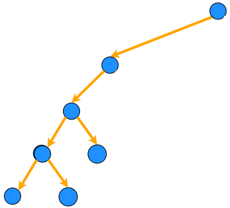


Path Exploration

input : a program P

output : a test suite TS covering all feasible paths of $Paths^{\leq k}(\texttt{P})$

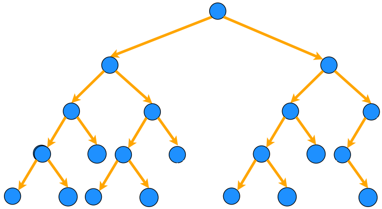
- pick a path $\sigma \in Paths^{\leq k}(\texttt{P})$
- compute a *path predicate* φ_σ of σ
- solve φ_σ for satisfiability
- SAT(s) ? get a new pair $< \text{s}, \sigma >$
- loop until no more path to cover



Path Exploration

input : a program P
output : a test suite TS covering all feasible paths of $Paths^{\leq k}(\mathtt{P})$

- pick a path $\sigma \in Paths^{\leq k}(\mathtt{P})$
- compute a *path predicate* φ_σ of σ
- solve φ_σ for satisfiability
- SAT(s) ? get a new pair $< s, \sigma >$
- loop until no more path to cover

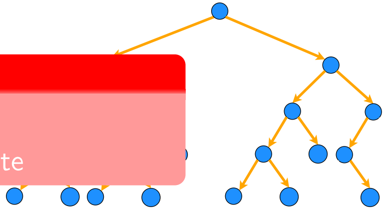


Path Exploration

input : a program P

output : a test suite *TS* covering all feasible paths of $Paths^{\leq k}(\texttt{P})$

- pick a path $\sigma \in Paths^{\leq k}(\texttt{P})$
- compute a *path predicate* φ_σ of σ
- solve φ_σ for satisfiability
- SAT(s) ? get a new pair $< s, \sigma >$
- loop until no more path to cover



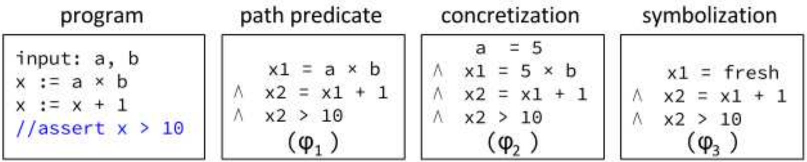
Beware

- ✗ #paths !
- ✗ incomplete

C/S for robustness and tradeoffs

Robustness : what if the instruction cannot be reasoned about ?

- missing code, self-modification
- hash functions, dynamic memory accesses, NLA operators



Solutions

- **Concretization** : replace by runtime value [lose completeness]
- **Symbolization** : replace by fresh variable [lose correctness]

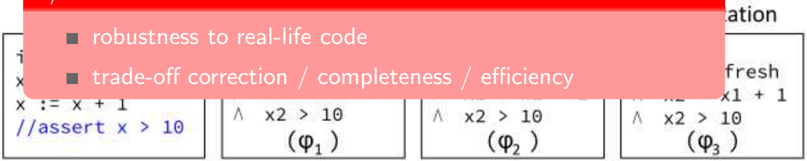
C/S for robustness and tradeoffs

Robustness : what if the instruction cannot be reasoned about ?

- missing code, self-modification
- hash functions, dynamic memory accesses, NLA operators

C/S essential to DSE

- robustness to real-life code
- trade-off correction / completeness / efficiency



Solutions

- **Concretization** : replace by runtime value [lose completeness]
- **Symbolization** : replace by fresh variable [lose correctness]

Outline

- about DSE
- the problem with C/S
- goal and results
- experiments
- conclusion

The problem with C/S policies

State of DSE

- Path predicate computation + solving ✓
- Path search : under active research
- **C/S : ? ? kind of black magic**

- hardcoded
- often a single C/S
- no easy tuning
- no reuse across tools

- undocumented, unclear
- many policies (one per tool)
- no comparison of C/S
- no systematic study of C/S

Unclear C/S policies

Consider the following situation

- instruction `x := @(a * b)`
- your tool documentation says : *"memory accesses are concretized"*
- suppose that at runtime : `a = 7`, `b = 3`

What is the intended meaning ? [perfect reasoning : $x == select(M, a \times b)$]

CS1 : $x == select(M, 21)$ [incorrect]

CS2 : $x == select(M, 21) \wedge a \times b == 21$ [minimal]

CS3 : $x == select(M, 21) \wedge a == 7 \wedge b == 3$ [atomic]

No best choice, depends on the context

- acceptable loss of correctness / completeness ?
- is it mandatory to get rid off $\times$?

Too many C/S policies

Just for C/S on memory accesses

- 4 basic policies : concretize or keep symbolic reads / writes
- exotic variations : multi-level dereferencement [exe], domain restriction [osmose], taint-based [s. heelan], dataflow-based [mayhem], etc.
- flavors of concretization : minimal, atomic, incorrect
- *all can be combined together*

Our goal

Establish C/S as a proper field of study

- what is a generic C/S ?
- how DSE can handle generic C/S ?
- identify tradeoffs, sweetspots, etc.

First step : a specification mechanism for C/S

- clear, non-ambiguous [documentation]
- tool independent [reuse, sharing, tuning]
- executable [input for tools]

Our goal

Establish C/S as a proper field of study

- what is a generic C/S ?
- how DSE can handle generic C/S ?
- identify

First step : a

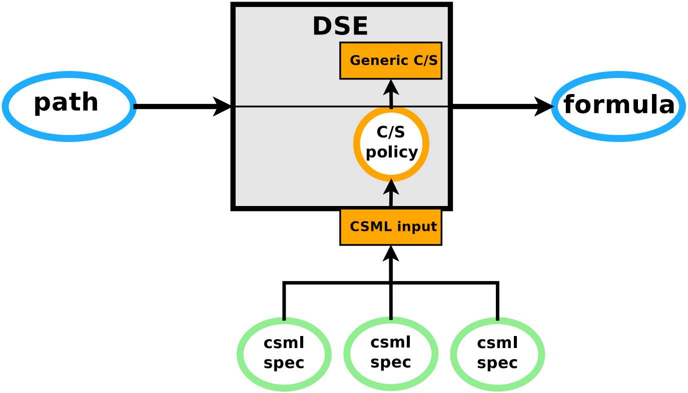
- clear, no … entation]
- tool ind … , tuning]
- executable … [input for tools]

Results

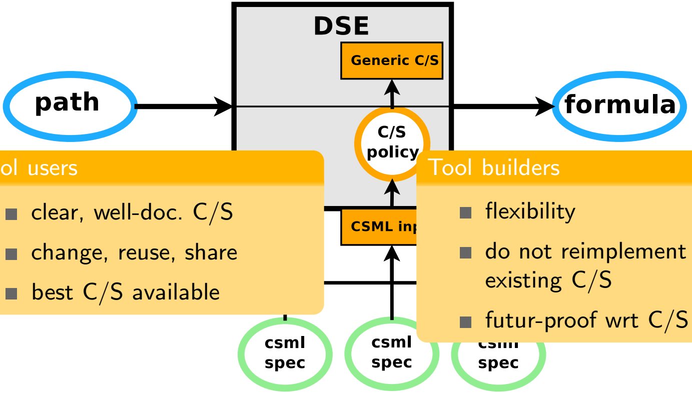
- formal definition of a generic C/S ✓
- a variant of DSE supporting generic C/S ✓
- CSML, a specification language for C/S ✓
- implementation in BINSEC ✓
- an experimental comparison of C/S policies ✓

Overview

Overview


DSE
Generic C/S
path
formula
C/S
policy
CSML input
csml
spec
csml
spec
csml
spec

Overview



Tool users

- clear, well-doc. C/S
- change, reuse, share
- best C/S available

Tool builders

- flexibility
- do not reimplement existing C/S
- futur-proof wrt C/S

What is a C/S policy ?

A decision function queried

- within path predicate computation
- before logical evaluation of an expression
- in the scope of a given location, instruction and memory state

$$\textsf{cs} : \textsf{loc} \times \textsf{instr} \times \textsf{state} \times \textsf{expr} \mapsto \left\{ \begin{array}{ll} \mathcal{C} & \textsf{concretization} \\ \mathcal{S} & \textsf{symbolization} \\ \mathcal{P} & \textsf{propagation} \end{array} \right\}$$

DSE with parametric C/S

Example :

- loc : `x := a + b`
- concrete memory state : $\{\mathtt{a} \mapsto 3; \mathtt{b} \mapsto 5\}$
- symbolic memory state : $\{\mathtt{a} \mapsto a_2; \mathtt{b} \mapsto b_9\}$

Standard evaluation, no C/S : $[\![\mathtt{a + b}]\!] \mapsto a_2 + b_9$

Evaluation with propagation : $[\![\mathtt{a + b}]\!]_{cs=\mathcal{P}} \mapsto (a_2 + b_9, \top)$

Evaluation with symbolization : $[\![\mathtt{a + b}]\!]_{cs=\mathcal{S}} \mapsto (\text{fresh}, \top)$

Evaluation with concretization : $[\![\mathtt{a + b}]\!]_{cs=\mathcal{C}} \mapsto (8, a_2 + b_9 = 8)$

CSML overview

Rule-based language $guard \Rightarrow \{\mathcal{C}, \mathcal{S}, \mathcal{P}\}$

Guard of the form $\pi_{loc} :: \pi_{ins} :: \pi_{expr} :: \pi_{\Sigma}$

- predicates on the location, instruction, expression, concrete memory state
- π_{ins} and π_{expr} mostly based on pattern matching and subterm checking
- predicates checked sequentially
- limited communication : *meta-variables* ($?x$, $?\star$) and *placeholders* ($!x$, $!_{\Box}$)

Set of rules

- checked sequentially, the first fireable rule returns
- presence of a default rule

Example of specifications (1)

$$\pi_{loc} :: \pi_{ins} :: \pi_{expr} :: \pi_{\Sigma} \Rightarrow \{\mathcal{C}, \mathcal{S}, \mathcal{P}\}$$

$*$	::	$*$	::	$\langle @?\star \rangle$	::	$*$	$\Rightarrow \mathcal{C}$;
default							$\Rightarrow \mathcal{P}$;

Meaning

- concretize result of a read value
- or : *"if we are evaluating an expression e built with @, then e is concretized, otherwise it is propagated."*

Examples

- `x := a + @b` : `@b` is concretized

Example of specifications (2)

$$\pi_{loc} :: \pi_{ins} :: \pi_{expr} :: \pi_{\Sigma} \Rightarrow \{\mathcal{C}, \mathcal{S}, \mathcal{P}\}$$

$*$	$::$	$\langle @?e := ?\star \rangle$	$::$	$\langle !e \rangle$	$::$	$*$	$\Rightarrow \mathcal{C}$;
default							$\Rightarrow \mathcal{P}$;

Meaning

- concretize write addresses
- or : *"if we are evaluating an expression e in the context of an assignment where e is used as the write address, then e is concretized, otherwise it is propagated."*

Examples

- `x := a + @b` : nothing is concretized
- `@x := a + @b` : `x` is concretized

Example of specifications (3)

$$\pi_{loc} :: \pi_{ins} :: \pi_{expr} :: \pi_{\Sigma} \Rightarrow \{\mathcal{C}, \mathcal{S}, \mathcal{P}\}$$

consider instruction `x := @(a * b)`, suppose at runtime : `a = 7`, `b = 3`

- minimal concretization of r/w expressions **[CS2]** [concretize `a*b`]

  $* :: \langle ?i \rangle :: (@\ !_{\square}) \prec !i :: * \Rightarrow \mathcal{C}$

- recursive concretization of r/w expressions : [concretize `a*b, a, b`]

  $* :: \langle ?i \rangle :: !_{\square} \prec (@\ ?\star) \prec !i :: * \Rightarrow \mathcal{C}$

- atomic concretization of r/w expressions **[CS3]** [concretize `a, b`]

  $* :: \langle ?i \rangle :: \texttt{var}(!_{\square}) \wedge !_{\square} \prec (@\ ?\star) \prec !i :: * \Rightarrow \mathcal{C}$

- incorrect concretization of r/w expressions **[CS1]** [replace `a*b` by 21]

  $* :: \langle ?i \rangle :: (@\ !_{\square}) \prec !i :: * \Rightarrow \mathcal{S}_{[eval_{\Sigma}(!_{\square})]}$

CSML good properties

Well-defined

- any CSML spec defines a C/S policy
- only $\mathcal{C}$ and $\mathcal{P}$: keeps correctness
- only $\mathcal{S}$ and $\mathcal{P}$: keeps completeness

Expressive enough

- sufficient for all examples from literature [systematic review]
- yet, still limited [say something about current C/S ?]

Implementable : see after

CSML good properties

Well-defined

- any CSML spec defines a C/S policy
- only $\mathcal{C}$ and $\mathcal{P}$: keeps correctness
- only $\mathcal{S}$ and $\mathcal{P}$: keeps completeness

Expressive enough

- sufficient for all examples from literature [systematic review]
- yet, still limited [say something about current C/S ?]

Implementable : see after

About the langage itself

- we describe the inner engine, not the user view
- syntax can be improved
- complexity can be hidden (predefined options, patterns)

Implementation and experiments

CSML implemented in BINSEC/SE [binary-level dse tool]

- first DSE tool with generic C/S support

Experiment 1 : evaluate CSML overhead

- vs : no C/S, C/S encoded via callbacks
- result : CSML does yield a cost, yet negligible wrt. solving time

Experiment 2 : experimental comparison of C/S policies

- five C/S policies for memory accesses : CC, CP, PC, PP*, PP
- result : PP* better on average, yet no clear winner : need different C/S !
- first time such a C/S comparison is performed !

CSML Overhead

Bench

- 167 programs (100 coreutils, 17 malware, 50 nist samate/verisec)
- $\approx$ 45,000 queries

		min	max	average
base	(PP)	0.04%	3%	0.3%
rule-based C/S policy	CC	0.1%	17%	1.2%
	CP	0.1%	23.5%	1.45%
	PC	0.08%	12.8%	0.85%
	PP*	0.08%	12.3%	0.95%
	PP	0.05%	4%	0.48%
hard-coded C/S policy	CC	0.05%	8.5%	0.5%
	CP	0.05%	8.2%	0.5%
	PC	0.05%	8%	0.45%
	PP*	0.05%	6%	0.45%
	PP	0.04%	3%	0.3%

Reported figures

- ratio between cost of formula creation and creation + solving
- note : solving time does not depend on the way C/S is implemented

Quantitative comparison

Five policies for memory accesses

- CC, PC, CP, PP*, PP
- first letter $\mapsto$ read operation, second letter $\mapsto$ write operation

	samate		core		malware		total	
	opt	best	opt	best	opt	best	opt	best
CC	20	0	44	1	5	0	69	1
PC	20	2	49	4	6	1	75	7
CP	23	1	61	11	4	0	88	12
PP*	36	12	71	24	10	5	**117**	**41**
PP	33	9	36	7	7	2	76	18

best (resp. opt) : number of programs for which the considered policy returns the strictly highest (resp. highest) number of SAT answers

Conclusion

Dynamic Symbolic Execution (DSE) : powerful approach to verif. and testing

- three key ingredients : path predicate computation & solving, path search, concretization & symbolization policy (C/S)

C/S is an essential part, yet mostly not studied

- many policies (one per tool), no systematic study of C/S
- undocumented, unclear
- tools : often a single hardcoded policy, no reuse across tools

Our goal : establish C/S as a proper field of study [focus first on specification]

- CSML, a specification language for C/S ✓
  - clear, non-ambiguous [documentation]
  - tool independent [reuse, sharing, tuning]
  - executable [input for tools]
- implemented in BINSEC ✓
- an experimental comparison of C/S policies ✓

Dynamic Symbolic Execution

Dynamic Symbolic Execution [Korel+, Williams+, Godefroid+]

- interleave dynamic and symbolic executions
- drive the search towards feasible paths for free
- give hints for relevant under-approximations [robustness]

Concretization : force a symbolic variable to take its runtime value

- application 1 : follow only feasible path for free
- application 2 : correct approximation of "difficult" constructs
  [out of scope or too expensive to handle]

About robustness

Goal = find input leading to ERROR
(assume we have only a solver for linear integer arith.)

```
f(int x, int y) {z=x*x; if (y == z) ERROR; else OK }
```

Loc	Instruction
0	input(x,y)
1	z := x * x
2	if (z == y) [True branch]

Path predicate (input X_0 et Y_0) — Unrealistic perfect symbolic reasoning
$\top$

About robustness

Goal = find input leading to ERROR

(assume we have only a solver for linear integer arith.)

```
f(int x, int y) {z=x*x; if (y == z) ERROR; else OK }
```

Loc	Instruction
0	input(x,y)
1	z := x * x
2	if (z == y) [True branch]

Path predicate (input X_0 et Y_0) — Unrealistic perfect symbolic reasoning

$\top \wedge Z_1 = X_0 \times X_0$

About robustness

Goal = find input leading to ERROR
(assume we have only a solver for linear integer arith.)

```
f(int x, int y) {z=x*x; if (y == z) ERROR; else OK }
```

Loc	Instruction
0	input(x,y)
1	z := x * x
2	if (z == y) [True branch]

Path predicate (input X_0 et Y_0) — Unrealistic perfect symbolic reasoning
$\top \wedge Z_1 = X_0 \times X_0 \wedge Z_1 = Y_0$

About robustness

Goal = find input leading to ERROR

(assume we have only a solver for linear integer arith.)

```
f(int x, int y) {z=x*x; if (y == z) ERROR; else OK }
```

Loc	Instruction
0	input(x,y)
1	z := x * x
2	if (z == y) [True branch]

Path predicate (input X_0 et Y_0) — Unrealistic perfect symbolic reasoning

OK, but how to solve ? ×

About robustness

Goal = find input leading to ERROR
(assume we have only a solver for linear integer arith.)

```
f(int x, int y) {z=x*x; if (y == z) ERROR; else OK }
```

Loc	Instruction
0	input(x,y)
1	z := x * x
2	if (z == y) [True branch]

Path predicate (input X_0 et Y_0) — Limited symbolic reasoning

$\top$

About robustness

Goal = find input leading to ERROR
(assume we have only a solver for linear integer arith.)

```
f(int x, int y) {z=x*x; if (y == z) ERROR; else OK }
```

Loc	Instruction
0	input(x,y)
1	z := x * x
2	if (z == y) [True branch]

Path predicate (input X_0 et Y_0) — Limited symbolic reasoning

$\top \wedge Z_1 = X_0 \times X_0$

About robustness

Goal = find input leading to ERROR

(assume we have only a solver for linear integer arith.)

```
f(int x, int y) {z=x*x; if (y == z) ERROR; else OK }
```

Loc	Instruction
0	input(x,y)
1	z := x * x
2	if (z == y) [True branch]

Path predicate (input X_0 et Y_0) — Limited symbolic reasoning

$\top \wedge \top$

About robustness

Goal = find input leading to ERROR

(assume we have only a solver for linear integer arith.)

```
f(int x, int y) {z=x*x; if (y == z) ERROR; else OK }
```

Loc	Instruction
0	input(x,y)
1	z := x * x
2	if (z == y) [True branch]

Path predicate (input X_0 et Y_0) — Limited symbolic reasoning

$\top \wedge \top \wedge Z_1 = Y_0$

About robustness

Goal = find input leading to ERROR
(assume we have only a solver for linear integer arith.)

```
f(int x, int y) {z=x*x; if (y == z) ERROR; else OK }
```

Loc	Instruction
0	input(x,y)
1	z := x * x
2	if (z == y) [True branch]

Path predicate (input X_0 et Y_0) — Limited symbolic reasoning
Incorrect, may find a bad solution (ex : $X_0 = 10,\ Y_0 = 34$) ×

About robustness

Goal = find input leading to ERROR
(assume we have only a solver for linear integer arith.)

```
f(int x, int y) {z=x*x; if (y == z) ERROR; else OK }
```

Loc	Instruction
0	input(x,y)
1	z := x * x
2	if (z == y) [True branch]

Path predicate (input X_0 et Y_0) — Limited dynamic symbolic reasoning
$\top$

About robustness

Goal = find input leading to ERROR

(assume we have only a solver for linear integer arith.)

```
f(int x, int y) {z=x*x; if (y == z) ERROR; else OK }
```

Loc	Instruction
0	input(x,y)
1	z := x * x
2	if (z == y) [True branch]

Path predicate (input X_0 et Y_0) — Limited dynamic symbolic reasoning

$\top \wedge Z_1 = X_0 \times X_0$ [assume runtime values : x=3,z=9]

About robustness

Goal = find input leading to ERROR

(assume we have only a solver for linear integer arith.)

```
f(int x, int y) {z=x*x; if (y == z) ERROR; else OK }
```

Loc	Instruction
0	input(x,y)
1	z := x * x
2	if (z == y) [True branch]

Path predicate (input X_0 et Y_0) — Limited dynamic symbolic reasoning

$\top \wedge Z_1 = 9 \wedge X_0 = 3$

About robustness

Goal = find input leading to `ERROR`

(assume we have only a solver for linear integer arith.)

```
f(int x, int y) {z=x*x; if (y == z) ERROR; else OK }
```

Loc	Instruction
0	input(x,y)
1	z := x * x
2	if (z == y) [True branch]

Path predicate (input X_0 et Y_0) — Limited dynamic symbolic reasoning

$\top \wedge Z_1 = 9 \wedge X_0 = 3 \wedge Z_1 = Y_0$

About robustness

Goal = find input leading to ERROR
(assume we have only a solver for linear integer arith.)

```
f(int x, int y) {z=x*x; if (y == z) ERROR; else OK }
```

Loc	Instruction
0	input(x,y)
1	z := x * x
2	if (z == y) [True branch]

Path predicate (input X_0 et Y_0) — Limited dynamic symbolic reasoning
Correct, find a real solution (ex : $X_0 = 3,\ Y_0 = 9$) ✓