Verified Symbolic Execution with Kripke Specification Monads

Steven Keuchel
Sander Huyghebaert
Georgy Lukyanov
Dominique Devriese

May 2nd, 2022

VUB SOFTWARE LANGUAGES LAB
Newcastle University
DistriNet KU LEUVEN

Flavours of program verifiers

Trusting

Hope

Dafny, Boogie, Why3, Viper, VCC, ESC/Java, ..

Skeptical

Witness producing

External:
HOL-Boogie

Meta-programming:
Bedrock1, Bedrock2, VST, CFML, F*, Iris, ...

Autarkic

Comput. reflection

MFVF, VeriSmall

Mixed flavours: skeptical first, autarkic steps

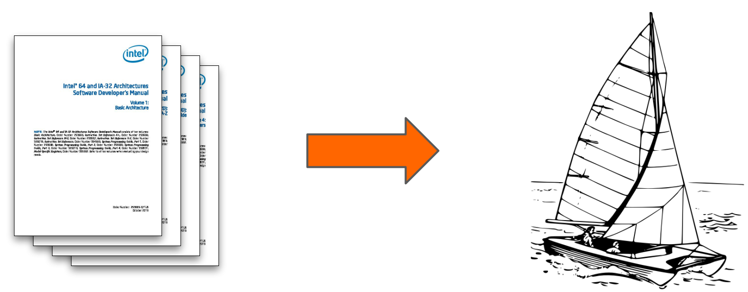
Challenges

- Complicated proofs
  - Deep-embedding of program logic assertions
  - Bookkeeping of logic variables
  - Soundness preserving transformations
- Practicality
  - Combinatorial explosion
  - Incompleteness
    - Undecidable (user) theory
    - Lack of verified solvers
  - Modularity

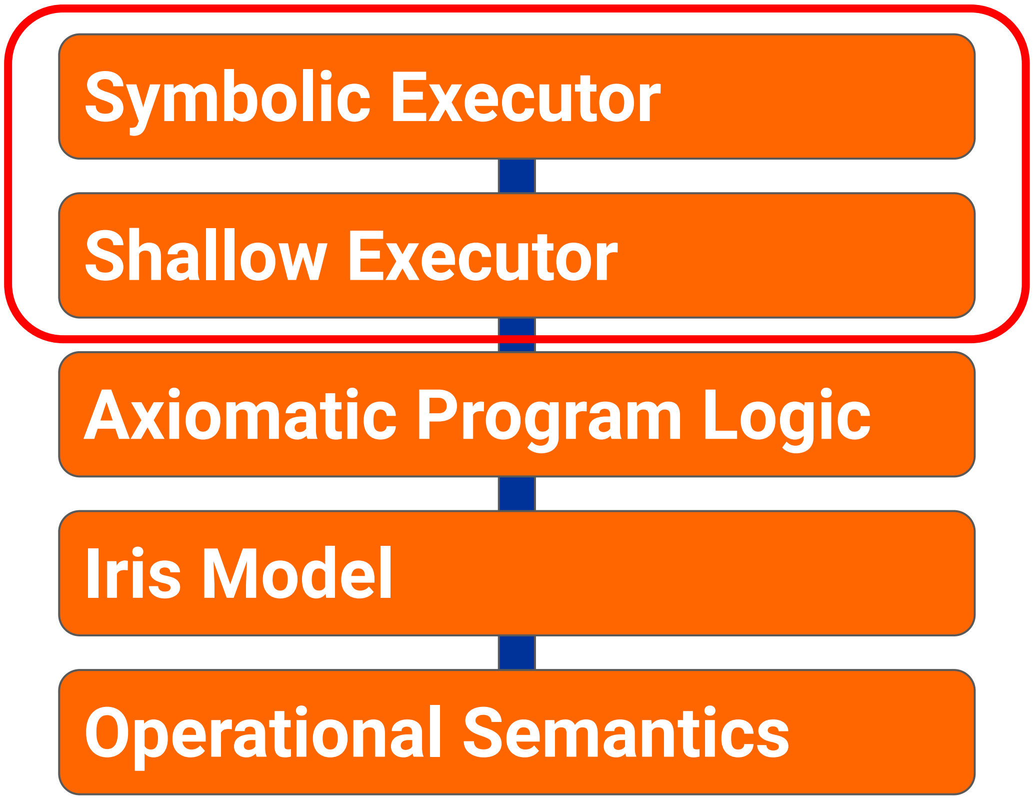
Contributions

- Systematic reusable approach for
  Semi-automatic symbolic execution-based program verifier with a machine-checked soundness proof.

- Implementation
  Katamaran Separation logic verifier for Sail



Katamaran


Symbolic Executor
Shallow Executor
Axiomatic Program Logic
Iris Model
Operational Semantics

Shallow Executor

Predicate Transformer Monads

Consider a weakest precondition

$$\mathtt{wp}(\mathtt{S}) \;:\; \mathtt{Pred}\ O \;\rightarrow\; \mathtt{Pred}\ I$$

taking $\mathtt{Pred} := \lambda\mathtt{x}.\mathtt{x}{\longrightarrow}\mathtt{Prop}$ $(\mathtt{or}\ \mathtt{iProp}\ \Sigma)$ and shuffling

$$\mathtt{wp}(\mathtt{S}) \;:\; I{\longrightarrow}\underbrace{(O{\longrightarrow}\mathtt{Prop}){\longrightarrow}\mathtt{Prop}}_{\mathtt{Cont\ Prop}\ O}$$

also backwards predicate transformer monads.

Specification Monads

- Can write specifications in the monad (F*-like)
  - Indexing (Dijkstra monads) or monad morphisms
  - Verification condition generator for monadic code
- Can write a monadic interpreter
  - Predicate transformer semantics
    for an object language
  - Verification condition generator
    for an object language
- (Add side effects with monad transformers)

Using meta-language eliminators

```
W := Cont ℙ
```

```
wp : exp -> W v
...
wp (IF e THEN e₁ ELSE e₂) :=
    v <- wp e; if v then wp e₁ else wp e₂

...
```

Propositional Features

```
W := Cont ℙ
```

- Angelic and demonic non-determinsm

```
angelic : W v                := λPOST. ∃v.POST v
demonic : W v                := λPOST. ∀v.POST v
_⊕_     : W A → W A → W A := λm₁ m₂ POST. m₁ POST ∨ m2 POST
_⊗_     : W A → W A → W A := λm₁ m₂ POST. m₁ POST ∧ m2 POST
```

- Guards

```
assert  : ℙ → W () := λQ POST. Q ∧ POST ()
assume  : ℙ → W () := λQ POST. Q → POST ()
consume : ℙ → W () := λQ POST. Q * POST ()
produce : ℙ → W () := λQ POST. Q -* POST ()
```

Avoid meta-language eliminators

```
W := Cont ℙ
```

```
wp : exp -> W v
...
wp (IF e THEN e₁ ELSE e₂) :=
  v <- wp e;
   (assume (v = true); wp e₁)
  ⊗(assume (v = false); wp e₂)

...
```

Symbolic Executor

Symbolic execution

- Define symbolic propositions

  ℓ ::= logic variable $\quad$ V ::= v | ℓ

  $\mathbb{F}$ ::= V = V | ...

  $\mathbb{S}$ ::= $\top$ | $\bot$ | $\mathbb{F} \rightarrow \mathbb{S}$ | $\mathbb{F} \wedge \mathbb{S}$ | $\mathbb{S} \wedge \mathbb{S}$

  $\quad$ | $\mathbb{S} \vee \mathbb{S}$ | $\exists \ell . \mathbb{S}$ | $\forall \ell . \mathbb{S}$
- Figure out fresh name generation (ℓ) and done?
  - Possible world semantics for dynamic logic variable allocation.
- Other concerns
  - Combinatorial explosion?

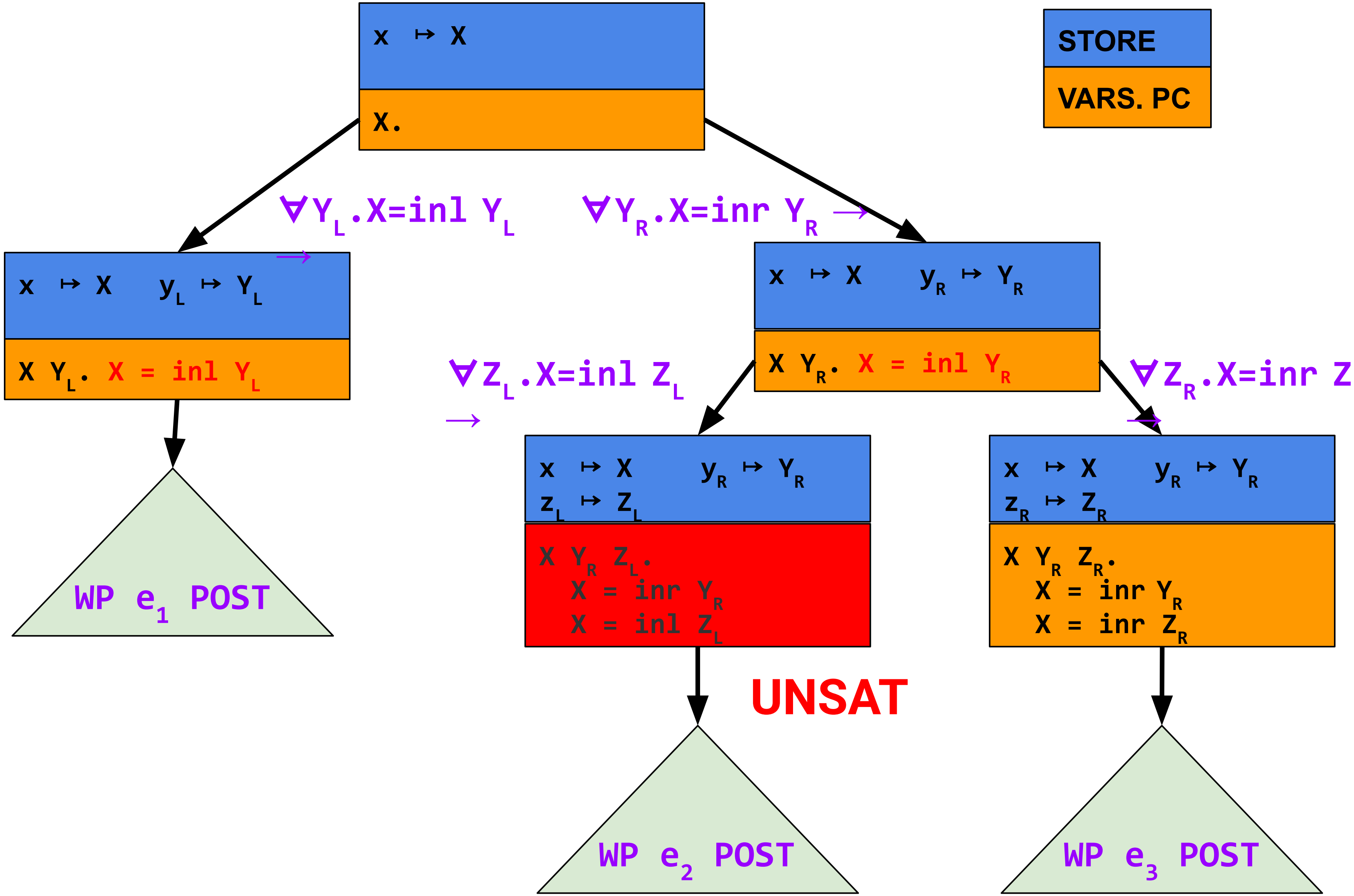
Avoiding path explosion

Symbolic executors

- explicitly represent past
  control-flow constraints (path constraints),
- algebraically simplify symbolic states,
- and eagerly prune unreachable cases
  (unsatisfiable constraints).

Example execution

```
WP (case x of
     | inl yl => e1
     | inr yr => case x of
                   | inl zl => e2
                   | inr zr => e3) POST
```



STORE
VARS. PC
x ↦ X
X.
∀Y_L.X=inl Y_L →
∀Y_R.X=inr Y_R →
x ↦ X y_L ↦ Y_L
X Y_L. X = inl Y_L
x ↦ X y_R ↦ Y_R
X Y_R. X = inl Y_R
∀Z_L.X=inl Z_L →
∀Z_R.X=inr Z_R →
x ↦ X y_R ↦ Y_R
z_L ↦ Z_L
X Y_R Z_L.
X = inr Y_R
X = inl Z_L
x ↦ X y_R ↦ Y_R
z_R ↦ Z_R
X Y_R Z_R.
X = inr Y_R
X = inr Z_R
WP e_1 POST
UNSAT
WP e_2 POST
WP e_3 POST

Symbolic execution reloaded

- Define worlds (contextual information)

$x \mapsto X \qquad y_R \mapsto Y_R$
$z_R \mapsto Z_R$

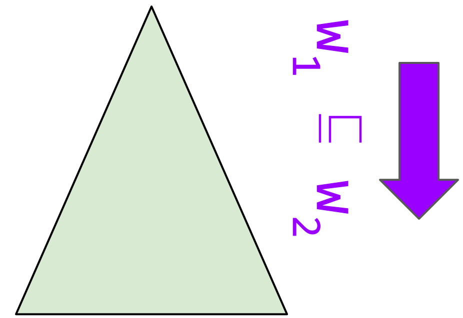
State

$X\ Y_R\ Z_R.$
$\quad X = \mathtt{inr}\ Y_R$
$\quad X = \mathtt{inr}\ Z_R$

World

- Work in ($\mathtt{World} \to \mathtt{Type}$), i.e. current world is always available in computations.
- View $\mathtt{V}, \mathbb{F}, \mathbb{S}$ as belonging to this category.

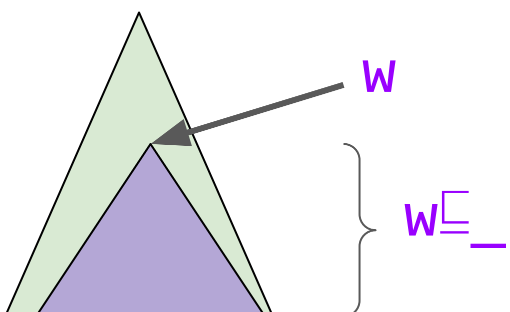
Symbolic propositions reloaded



- Define accessibility $w_1 \sqsubseteq w_2$

- Define box

$$(\square A)\ w := \forall\ w'.\ w \sqsubseteq w' \rightarrow A\ w'$$



- Symbolic execution monad

$$M\ A := \square(A \rightarrow \mathbb{S}) \rightarrow \mathbb{S}$$

Symbolic execution

```
M A := □(A→𝕊)→𝕊

WP : exp → ⊢ M V

...

WP (IF e THEN e₁ ELSE e₂) :=
  [ω] V <- wp e;
    (assume (V = true) ;[ω₁] WP e₁)
   ⊗(assume (V = false) ;[ω₂] WP e₂)

...
```

Pruning

```
M A := □(A→𝕊)→𝕊

ASSUME : ⊢ 𝔽 → M () :=
  λw f (POST : □(() → 𝕊) w).
    match solver w f with
    | Some f’ ⇒ let w’ := ... in
                let ω := ... in
                f’ → POST w’ ω ()

   | None ⇒ ⊤

  ...
```

Symbolic execution soundness

Refinement logical relation

$$\mathcal{R}_{\lesssim}[\![A, a]\!] \subseteq \{(w, \iota_w, A, a)\}$$

$$\mathcal{R}_{\lesssim}[\![\mathrm{V}, \mathrm{v}]\!] = \{(w, \iota_w, V, v) \mid v = V[\iota_w]\}$$

$$\mathcal{R}_{\lesssim}[\![\mathbb{S}, \mathbb{P}]\!] = \{(w, \iota, \mathbb{S}, \mathbb{P}) \mid (\iota \models \mathbb{S}) \Rightarrow \mathbb{P}\}$$

$$\mathcal{R}_{\lesssim}[\![\Box A, a]\!] = \{(w, \iota, A, a) \mid \forall w', \omega : w \sqsubseteq w', \iota'. \iota = \iota' \circ \omega \rightarrow (w', \iota', A, a) \in \mathcal{R}_{\lesssim}[\![A, a]\!]\}$$

$$\mathcal{R}_{\lesssim}[\![A \rightarrow B, a \rightarrow b]\!] = \{(w, \iota, f_s, f_c) \mid \forall (w, \iota, v_s, v_c) \in \mathcal{R}_{\lesssim}[\![A, a]\!]. (w, \iota, f_s\ v_s, f_c\ v_c) \in \mathcal{R}_{\lesssim}[\![B, b]\!]\}$$

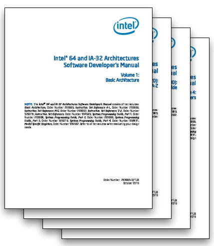
Soundness

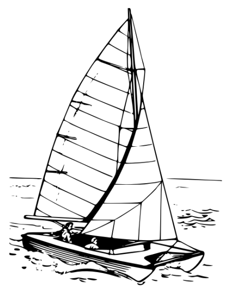
$$(w, \iota_w, \mathrm{WP}\ e, \mathrm{wp}\ e) \in \mathcal{R}_{\lesssim}[\![M\ V, W\ v]\!]$$

Katamaran

Katamaran

- Separation logic verifier for Sail (µSail),
  a domain specific language for specifying ISAs.





Semi-automatic

- Incomplete generic solver
  User-defined solver (pure)
- Ghost statements (spatial)
- Residual verification conditions
- Compose with proofs in IPM

ISA security

- Prove universal contract for arbitrary (adversarial) code

  { SECPRE } fetch-decode-execute-loop { SECPOST }
- Few interesting cases
  - Low-level memory access
  - Permissions checks, ...
- Aggressive over-approximation, e.g.

  `{ GPRS } func { GPRS }`
- Lots of boilerplate
  - Most functions are not security critical
  - 10k - 100k of code

Case study - MinimalCaps

	µSail functions (Katamaran)	Foreign functions (IPM)
Source	441 LoC	12 LoC
Verification	0.415s	12.1s
Proof	1 LoC	220 LoC

Case study - RISC-V PMP

	µSail functions (Katamaran)	µSail functions (IPM)	Foreign functions (IPM)
Source	567 LoC	1 LoC	8 LoC
Verification	< 2min	??	??
Proof	40 LoC	??	??

Comparison - Singly-linked lists

	Katamaran				Bedr.	VST	SLF
	Symbolic VC			Solver			
	Branches	Pruned	Time	Time	Time	Time	Time
append	2	0	0.0075	–	31.5	2.61	–
$\text{append}_{\text{loop}}$	3	1	0.033	–			0.99
copy	3	1	0.018	–	–	–	0.95
length	3	1	0.022	0.15	16.8	–	0.78
reverse	1	0	0.0037	–	20.0	2.34	–
$\text{reverse}_{\text{loop}}$	3	1	0.026	0.25			–
summaxlen	3	0	3.28	–	–	–	–
Lemmas			0.22		1.05	–	0.33

Future Work

- VCs in separation logic
- Known assembly code verification
- Pure automation
  - Linear bitvector theory
- Spatial automation
  - User provided solvers
  - Custom tactic language
- Reusability

Thanks for your Attention!

https://katamaran-project.github.io/
https://github.com/katamaran-project/katamaran