Gillian

Compositional Symbolic Testing and Verification

PHILIPPA GARDNER

Imperial College London

Petar Maksimović

José Fragoso Santos

Sacha-Élie Ayoun

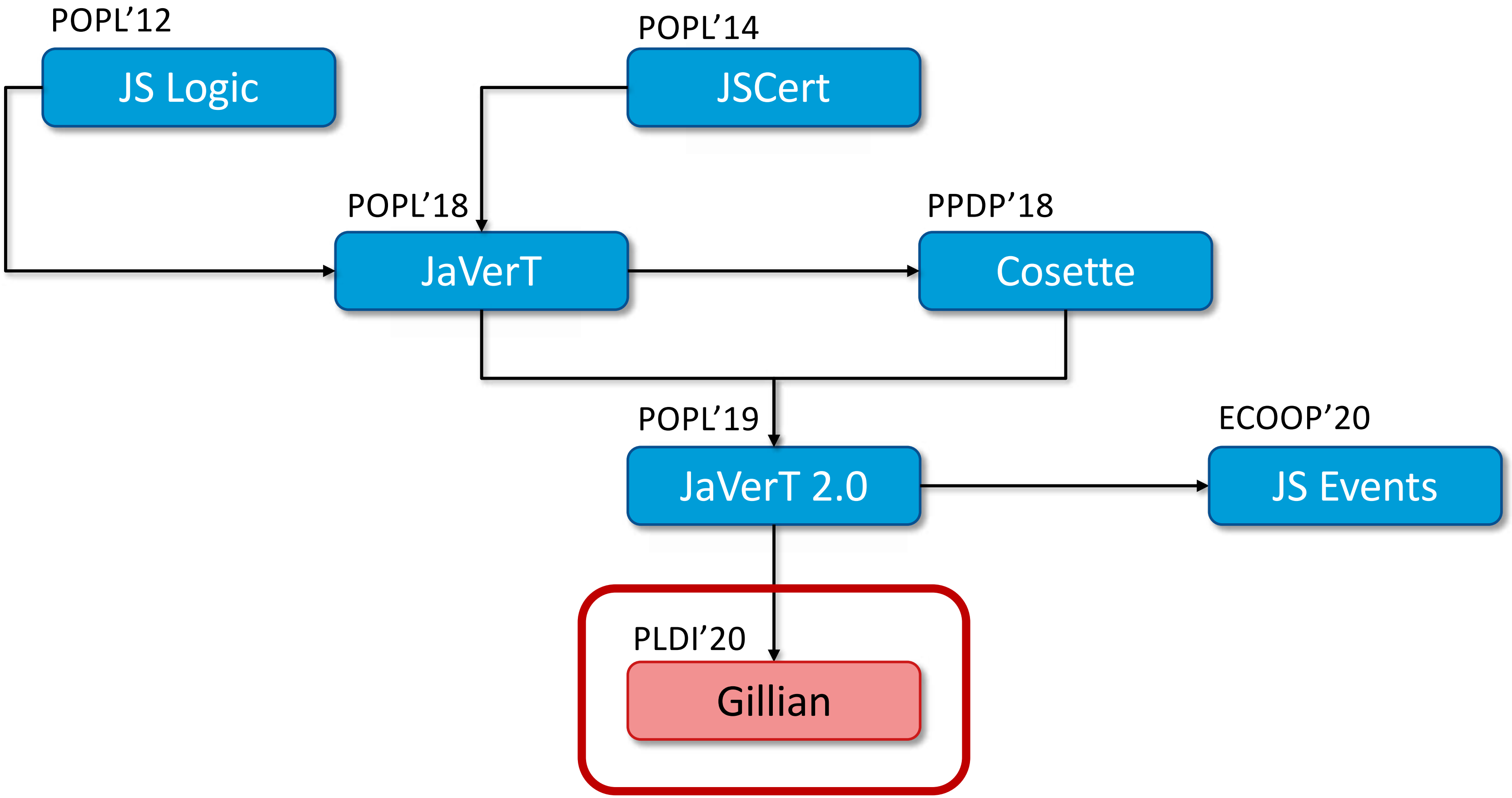
Philippa Gardner

Imperial College London TÉCNICO LISBOA inesc id lisboa

WITH GENEROUS SUPPORT FROM

amazon EPSRC Engineering and Physical Sciences Research Council facebook FCT Fundação para a Ciência e a Tecnologia National Cyber Security Centre a part of GCHQ

Verified Software: JavaScript Analysis



POPL'12
JS Logic

POPL'14
JSCert

POPL'18
JaVerT

PPDP'18
Cosette

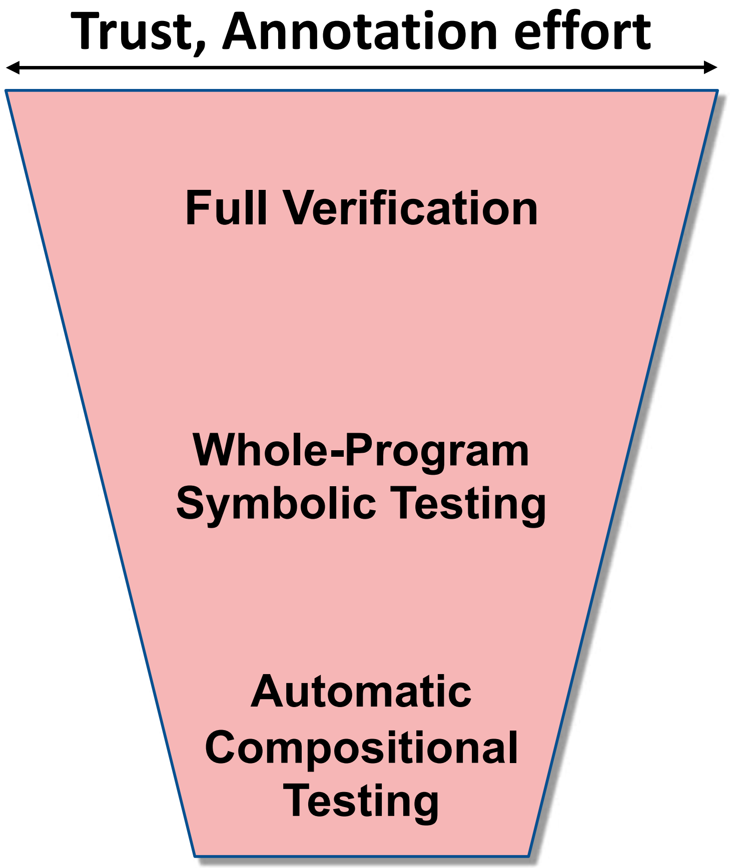
POPL'19
JaVerT 2.0

ECOOP'20
JS Events

PLDI'20
Gillian

Gillian: Unified Symbolic Analysis

■ Technique 🟦 Annotations 🟩 Success 🟥 Failure



Trust, Annotation effort

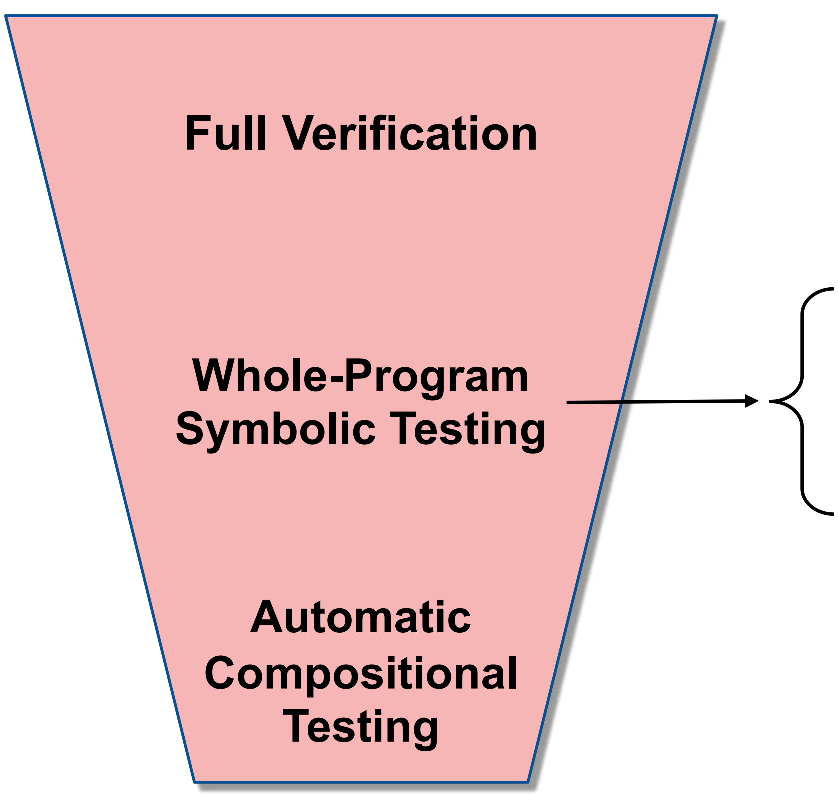
Full Verification

Whole-Program Symbolic Testing

Automatic Compositional Testing

Gillian: Unified Symbolic Analysis

■ Technique 🟦 Annotations 🟩 Success 🟥 Failure



Trust, Annotation effort

Full Verification

Whole-Program Symbolic Testing

Automatic Compositional Testing

Core Execution engine (Gillian: Part 1, PLDI'20)

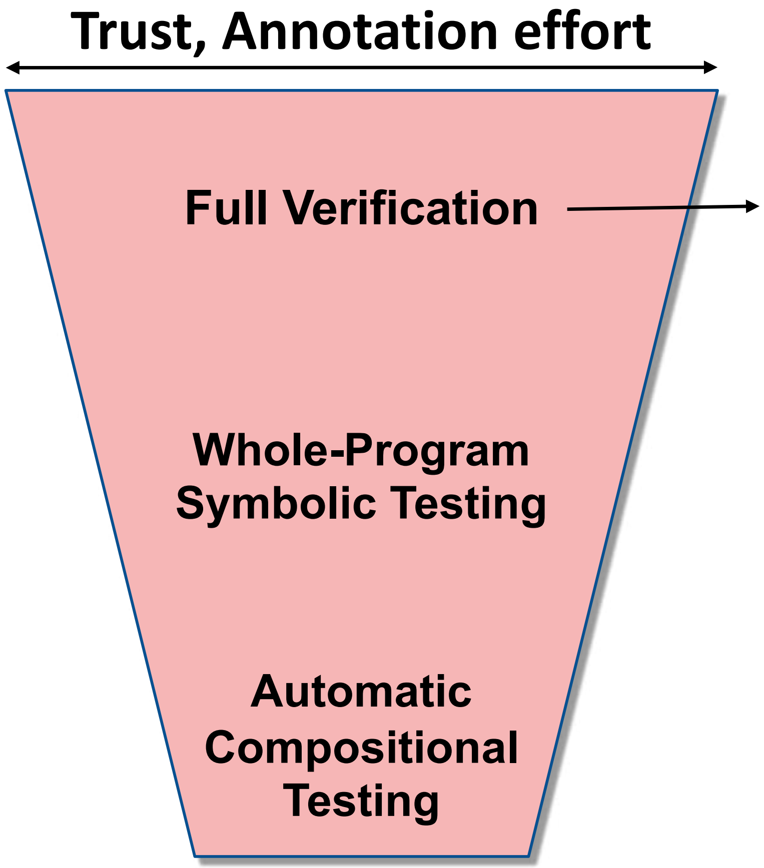
First-order assertions about symbolic inputs/outputs

Bounded correctness guarantees (loops unrolled)

Counter-models

Gillian: Unified Symbolic Analysis

⬛ Technique 🟦 Annotations 🟩 Success 🟥 Failure



Specification (Gillian Verification for JS and C, submitted)

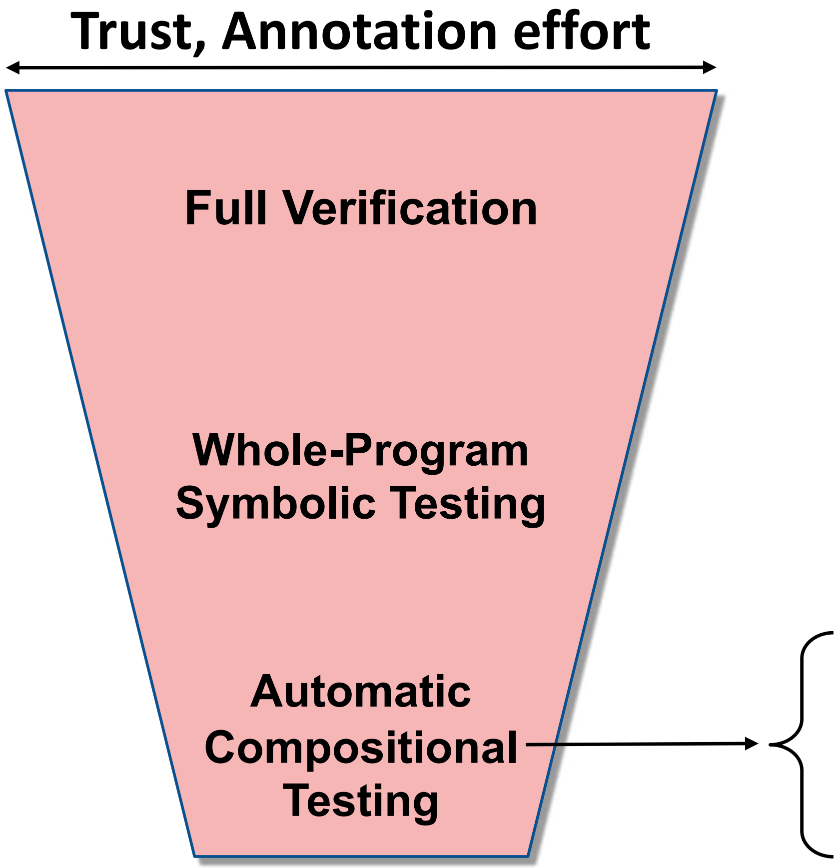
Separation-logic specs, invariants, lemmas, tactics

Functional correctness guarantees

Failing symbolic traces

Gillian: Unified Symbolic Analysis

■ Technique 🟦 Annotations 🟩 Success 🟥 Failure



Trust, Annotation effort

Full Verification

Whole-Program Symbolic Testing

Automatic Compositional Testing

Bi-abduction

None / Lightweight type annotations

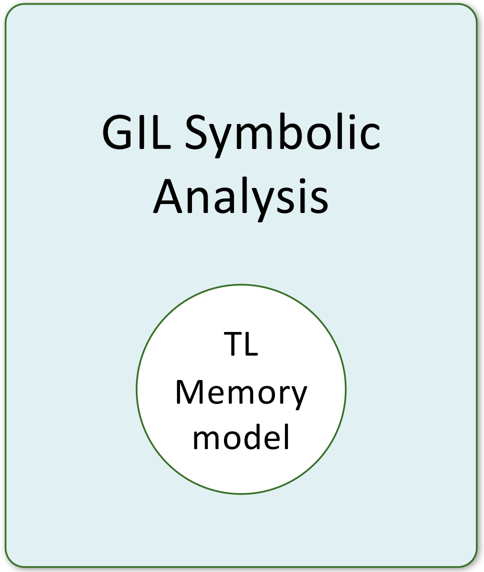
Bounded correctness specifications (loops unrolled)

Lightweight bug specifications

The Gillian Platform

GILLIAN

The Gillian Platform

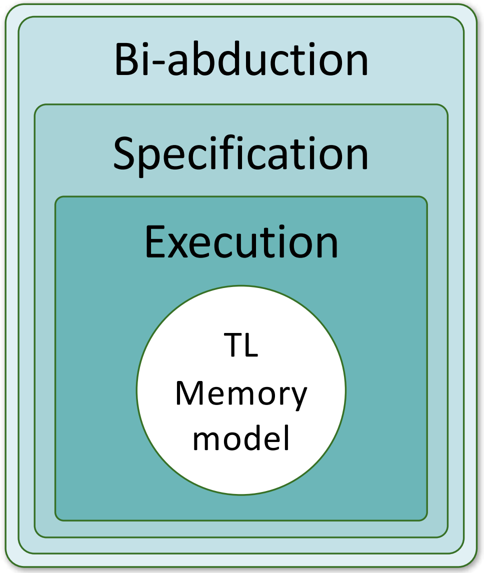

GIL Symbolic Analysis

TL Memory model

Gillian Infrastructure

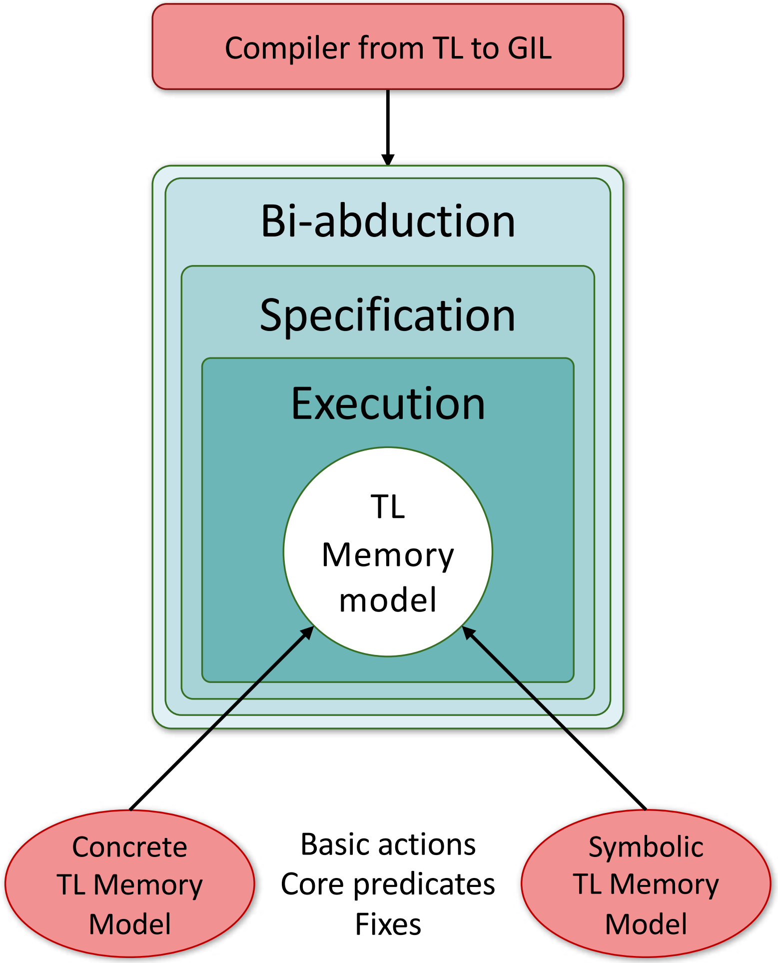
- GIL, an intermediate goto language parametric on the memory model of the target language (TL)
- First-order solver powered by the Z3 theorem prover

The Gillian Platform



Gillian Infrastructure

- GIL, an intermediate goto language parametric on the memory model of the target language (TL)
- First-order solver powered by the Z3 theorem prover
- Modular analyses: execution, specification, bi-abduction



The Gillian Platform

Gillian Instantiation (by a tool developer)

- OCaml impl. of TL **concrete** and **symbolic** memory models, using **basic actions**, **core predicates** and **fixes**
- Trusted **compiler** from the TL to GIL, preserving the memory models and the semantics

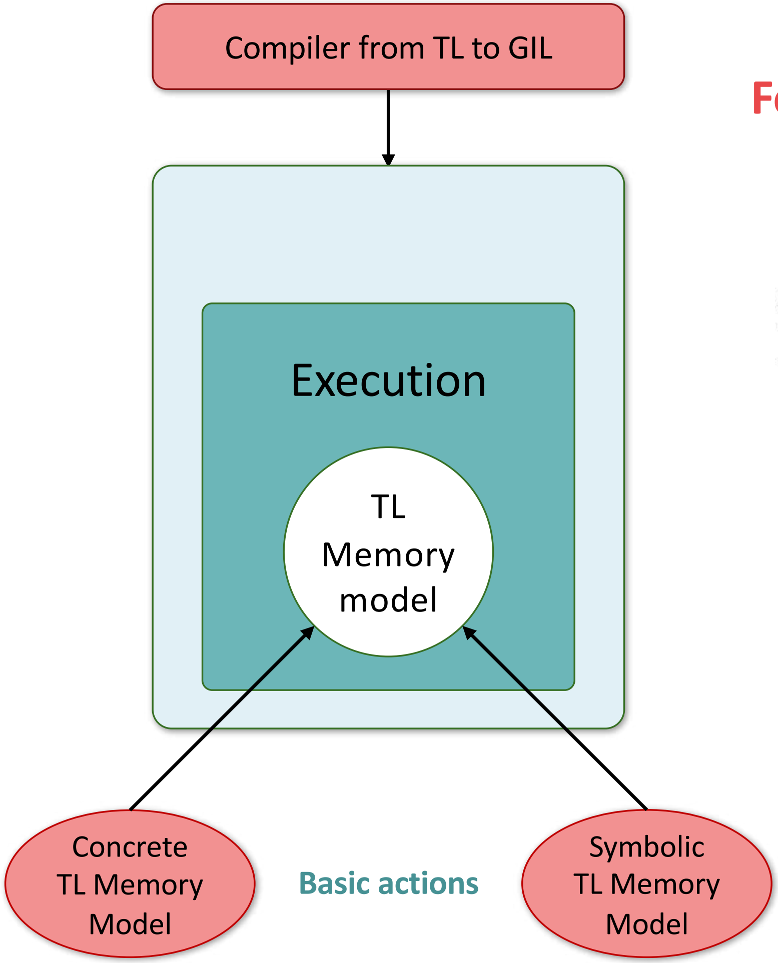
Example Instantiations

- **Gillian-While:** for teaching and experimentation
- **Gillian-JS:** extensible-object memory model, JaVerT compiler
- **Gillian-C:** block-offset memory model, CompCert compiler
- **Gillian-Rust:** just started (Sacha)

Gillian

IN THEORY

Core Execution Engine



Compiler from TL to GIL

Execution

TL Memory model

Concrete TL Memory Model

Basic actions

Symbolic TL Memory Model

Formal semantics, closely followed by OCaml implementation

$$\textsc{Assignment}\quad \frac{\mathsf{cmd}(\mathsf{p}, cs, i) = x := e \qquad \sigma.(\mathrm{setVar}_x \circ \mathrm{eval}_e)(-) \leadsto \sigma'}{\mathsf{p} \vdash \langle \sigma, cs, i \rangle \leadsto \langle \sigma', cs, i{+}1 \rangle}$$

$$\textsc{IfGoto - True}\quad \frac{\mathsf{cmd}(\mathsf{p}, cs, i) = \mathrm{ifgoto}\ e\ j \qquad \sigma.(\mathrm{assume} \circ \mathrm{eval}_e)(-) \leadsto \sigma'}{\mathsf{p} \vdash \langle \sigma, cs, i \rangle \leadsto \langle \sigma', cs, j \rangle}$$

$$\textsc{IfGoto - False}\quad \frac{\mathsf{cmd}(\mathsf{p}, cs, i) = \mathrm{ifgoto}\ e\ j \qquad \sigma.(\mathrm{assume} \circ \mathrm{eval}_{\neg e})(-) \leadsto \sigma'}{\mathsf{p} \vdash \langle \sigma, cs, i \rangle \leadsto \langle \sigma', cs, i+1 \rangle}$$

$$\textsc{Call}\quad \frac{\begin{array}{c}\mathsf{cmd}(\mathsf{p}, cs, i) = x := e(e') \quad \sigma.((\mathrm{getStore} \mathbin{\bar{\circ}} \mathrm{eval}_{e'}) \mathbin{\bar{\circ}} \mathrm{eval}_e)([-,[-,-]]) \leadsto (\sigma', [f, [\mathsf{v}, \rho']]) \\ cs' = \langle f, x, \rho', i{+}1 \rangle : cs \quad \mathsf{p}(f) = f(y)\{\overline{c}\} \quad \sigma'.\mathrm{setStore}([[y, \mathsf{v}]]) \leadsto \sigma''\end{array}}{\mathsf{p} \vdash \langle \sigma, cs, i \rangle \leadsto \langle \sigma'', cs', 0 \rangle}$$

$$\textsc{Return}\quad \frac{\begin{array}{c}\mathsf{cmd}(\mathsf{p}, cs, i) = \mathrm{return}\ e \quad cs = \langle -, x, \rho, j \rangle : cs' \\ \sigma.\mathrm{eval}_e(-) \leadsto (\sigma', \mathsf{v}) \quad \sigma'.(\mathrm{setVar}_x \mathbin{\bar{\circ}} \mathrm{setStore})([\rho, \mathsf{v}]) \leadsto \sigma''\end{array}}{\mathsf{p} \vdash \langle \sigma, cs, i \rangle \leadsto \langle \sigma'', cs', j \rangle}$$

$$\textsc{Top Return}\quad \frac{\begin{array}{c}\mathsf{cmd}(\mathsf{p}, cs, i) = \mathrm{return}\ e \\ cs = \langle f \rangle \quad \sigma.\mathrm{eval}_e(-) \leadsto (\sigma', \mathsf{v})\end{array}}{\mathsf{p} \vdash \langle \sigma, cs, i \rangle \leadsto \langle \sigma', cs, i \rangle^{\mathsf{N(v)}}}$$

$$\textsc{Fail}\quad \frac{\begin{array}{c}\mathsf{cmd}(\mathsf{p}, cs, i) = \mathrm{fail}\ e \\ \sigma.\mathrm{eval}_e(-) \leadsto (\sigma', \mathsf{v})\end{array}}{\mathsf{p} \vdash \langle \sigma, cs, i \rangle \leadsto \langle \sigma', cs, i \rangle^{\mathsf{E(v)}}}$$

$$\textsc{uSym}\quad \frac{\begin{array}{c}\mathsf{cmd}(\mathsf{p}, cs, i) = x := \mathrm{uSym}_j \\ \sigma.(\mathrm{setVar}_x \circ \mathrm{uSym})(j) \leadsto \sigma'\end{array}}{\mathsf{p} \vdash \langle \sigma, cs, i \rangle \leadsto \langle \sigma', cs, i{+}1 \rangle}$$

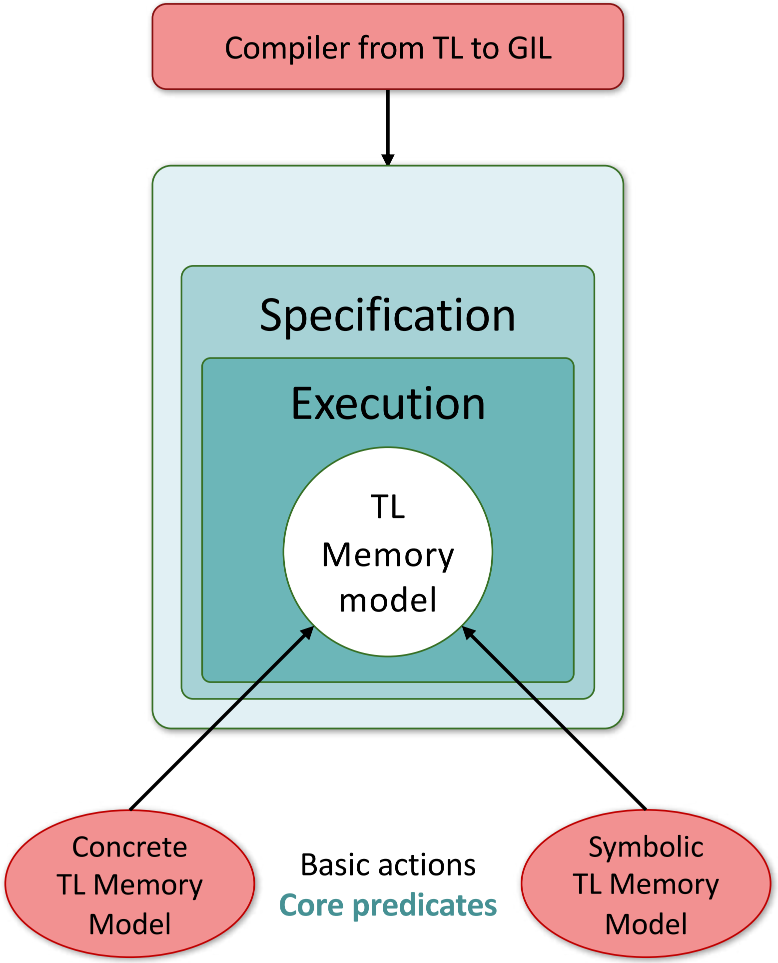
$$\textsc{iSym}\quad \frac{\begin{array}{c}\mathsf{cmd}(\mathsf{p}, cs, i) = x := \mathrm{iSym}_j \\ \sigma.(\mathrm{setVar}_x \circ \mathrm{iSym})(j) \leadsto \sigma'\end{array}}{\mathsf{p} \vdash \langle \sigma, cs, i \rangle \leadsto \langle \sigma', cs, i{+}1 \rangle}$$

User Input: Basic Actions

Fundamental interactions between
the language and its memory

Single Additional Rule

$$\textsc{Action}\quad \frac{\begin{array}{c}\mathsf{cmd}(\mathsf{p}, cs, i) = x := \alpha(e) \\ \sigma.(\mathrm{setVar}_x \circ \alpha \circ \mathrm{eval}_e)(-) \leadsto \sigma'\end{array}}{\mathsf{p} \vdash \langle \sigma, cs, i \rangle \leadsto \langle \sigma', cs, i{+}1 \rangle}$$



Compiler from TL to GIL

Specification

Execution

TL
Memory
model

Concrete
TL Memory
Model

Basic actions
Core predicates

Symbolic
TL Memory
Model

Specification

User Input: Core predicates

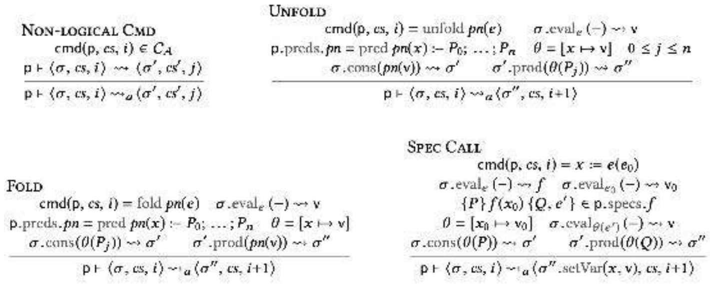
Separation-logic assertions
describing the memory building blocks

Consumer and **producer** actions
for each core predicate

Additional Rules

Unfolding/folding of **user-defined predicates**

Re-use of function specifications



NON-LOGICAL CMD

$$\frac{\begin{array}{c}\mathrm{cmd}(\mathsf{p}, cs, i) \in C_{\mathcal{A}} \\ \mathsf{p} \vdash \langle \sigma, cs, i \rangle \rightsquigarrow \langle \sigma', cs', j \rangle\end{array}}{\mathsf{p} \vdash \langle \sigma, cs, i \rangle \rightsquigarrow_{a} \langle \sigma', cs', j \rangle}$$

UNFOLD

$$\frac{\begin{array}{c}\mathrm{cmd}(\mathsf{p}, cs, i) = \mathrm{unfold}\ pn(e) \quad \sigma.\mathrm{eval}_{e}(-) \rightsquigarrow \mathsf{v} \\ \mathsf{p}.\mathrm{preds}.pn = \mathrm{pred}\ pn(x) :- P_0; \ldots; P_n \quad \theta = [x \mapsto \mathsf{v}] \quad 0 \le j \le n \\ \sigma.\mathrm{cons}(pn(\mathsf{v})) \rightsquigarrow \sigma' \quad \sigma'.\mathrm{prod}(\theta(P_j)) \rightsquigarrow \sigma''\end{array}}{\mathsf{p} \vdash \langle \sigma, cs, i \rangle \rightsquigarrow_{a} \langle \sigma'', cs, i+1 \rangle}$$

FOLD

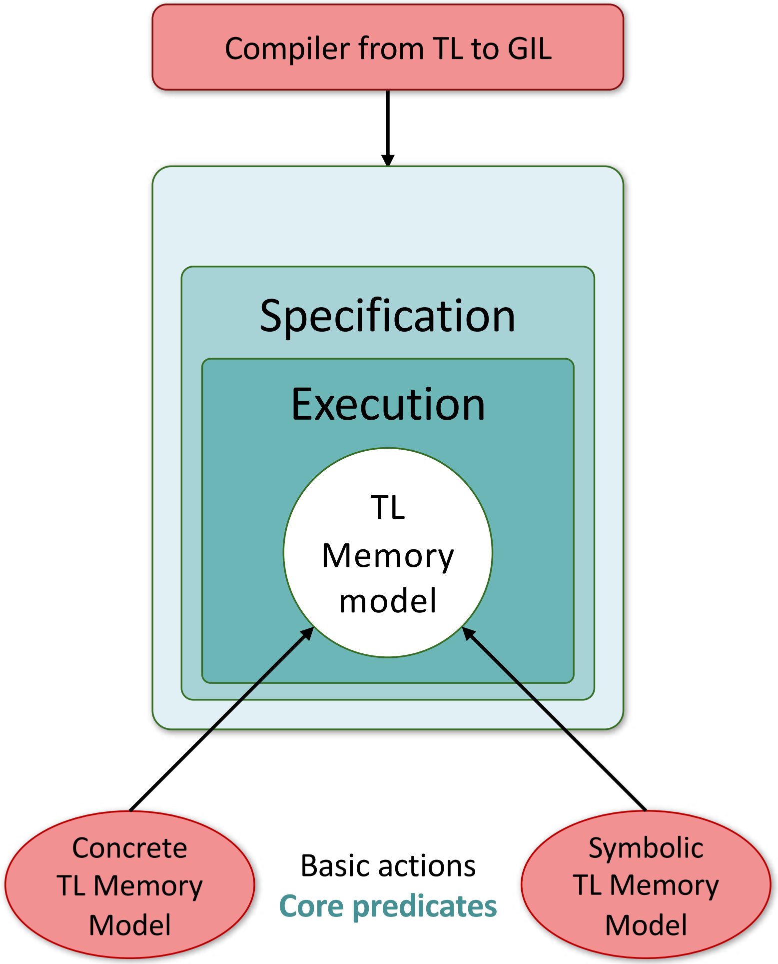
$$\frac{\begin{array}{c}\mathrm{cmd}(\mathsf{p}, cs, i) = \mathrm{fold}\ pn(e) \quad \sigma.\mathrm{eval}_{e}(-) \rightsquigarrow \mathsf{v} \\ \mathsf{p}.\mathrm{preds}.pn = \mathrm{pred}\ pn(x) :- P_0; \ldots; P_n \quad \theta = [x \mapsto \mathsf{v}] \\ \sigma.\mathrm{cons}(\theta(P_j)) \rightsquigarrow \sigma' \quad \sigma'.\mathrm{prod}(pn(\mathsf{v})) \rightsquigarrow \sigma''\end{array}}{\mathsf{p} \vdash \langle \sigma, cs, i \rangle \rightsquigarrow_{a} \langle \sigma'', cs, i+1 \rangle}$$

SPEC CALL

$$\frac{\begin{array}{c}\mathrm{cmd}(\mathsf{p}, cs, i) = x := e(e_0) \\ \sigma.\mathrm{eval}_{e}(-) \rightsquigarrow f \quad \sigma.\mathrm{eval}_{e_0}(-) \rightsquigarrow \mathsf{v}_0 \\ \{P\}\, f(x_0)\, \{Q, e'\} \in \mathsf{p}.\mathrm{specs}.f \\ \theta = [x_0 \mapsto \mathsf{v}_0] \quad \sigma.\mathrm{eval}_{\theta(e')}(-) \rightsquigarrow \mathsf{v} \\ \sigma.\mathrm{cons}(\theta(P)) \rightsquigarrow \sigma' \quad \sigma'.\mathrm{prod}(\theta(Q)) \rightsquigarrow \sigma''\end{array}}{\mathsf{p} \vdash \langle \sigma, cs, i \rangle \rightsquigarrow_{a} \langle \sigma''.\mathrm{setVar}(x, \mathsf{v}), cs, i+1 \rangle}$$

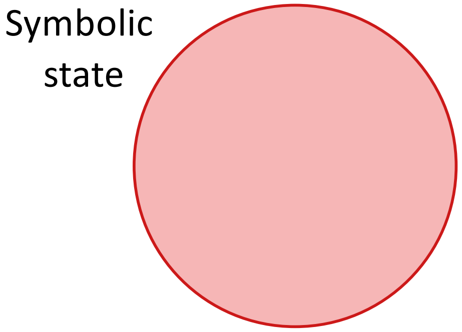
(slightly simplified)

Aside: Specification Re-Use



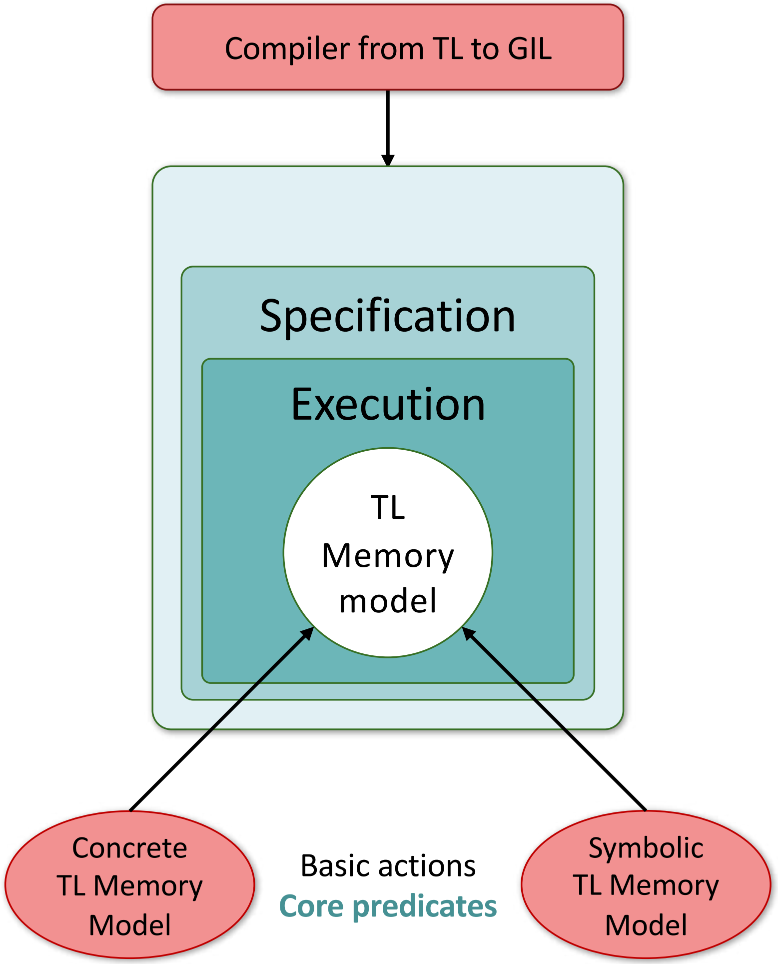
Function specifications: $\{ P \} f(x) \{ Q \}$

Goal: apply a given function specification instead of symbolically executing a function



$\{ P \} f(x) \{ Q \}$

Specification


Compiler from TL to GIL

Specification

Execution

TL Memory model

Concrete TL Memory Model

Basic actions

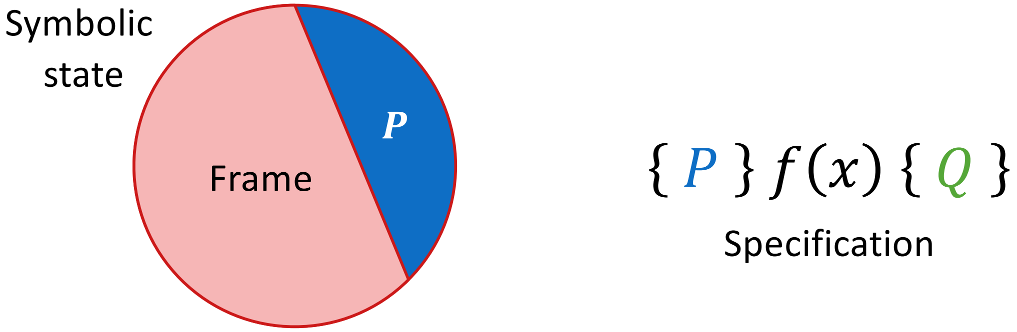
Core predicates

Symbolic TL Memory Model

Aside: Specification Re-Use

Function specifications: $\{\, P \,\}\, f(x)\, \{\, Q \,\}$

Goal: apply a given function specification instead of symbolically executing a function



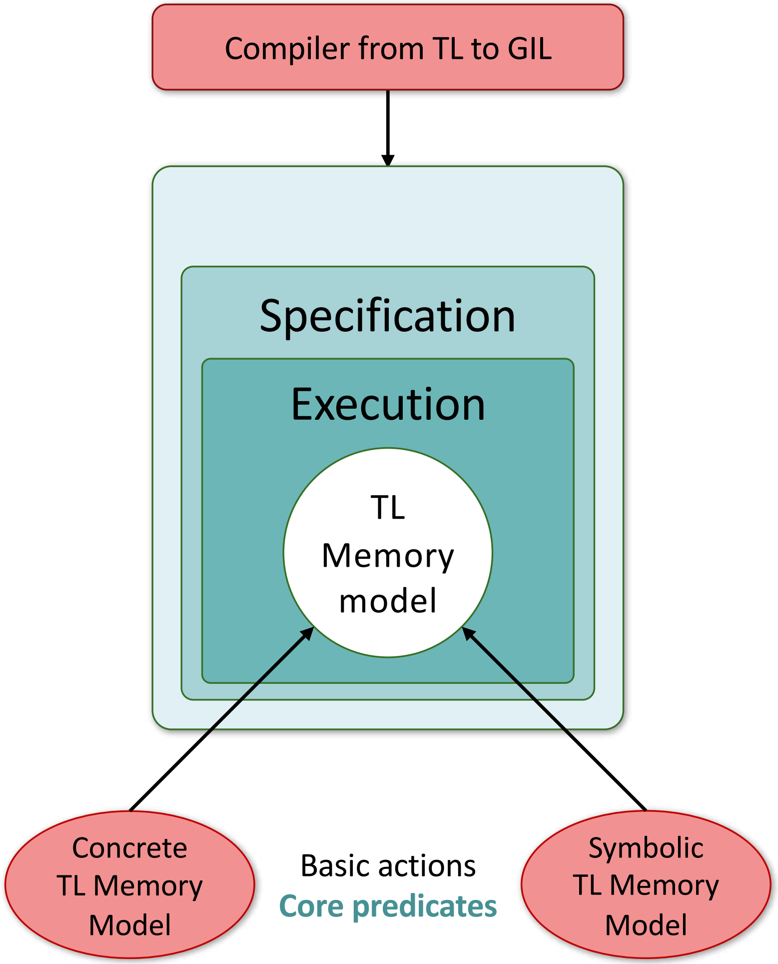
$\{\, P \,\}\, f(x)\, \{\, Q \,\}$

Specification

Step 1: consume the pre-condition

Unify the part of the state that corresponds to the pre-condition and consume it, leaving the frame; learn the bindings θ for the logical variables in the pre-condition

Aside: Specification Re-Use



Compiler from TL to GIL

Specification

Execution

TL Memory model

Concrete TL Memory Model

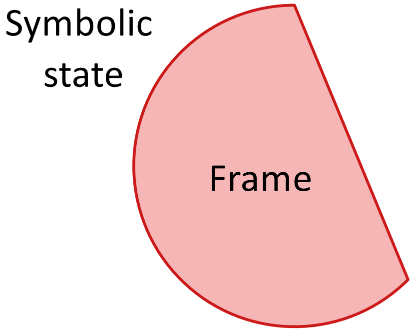
Basic actions

Core predicates

Symbolic TL Memory Model

Function specifications: $\{ P \} f(x) \{ Q \}$

Goal: apply a given function specification instead of symbolically executing a function



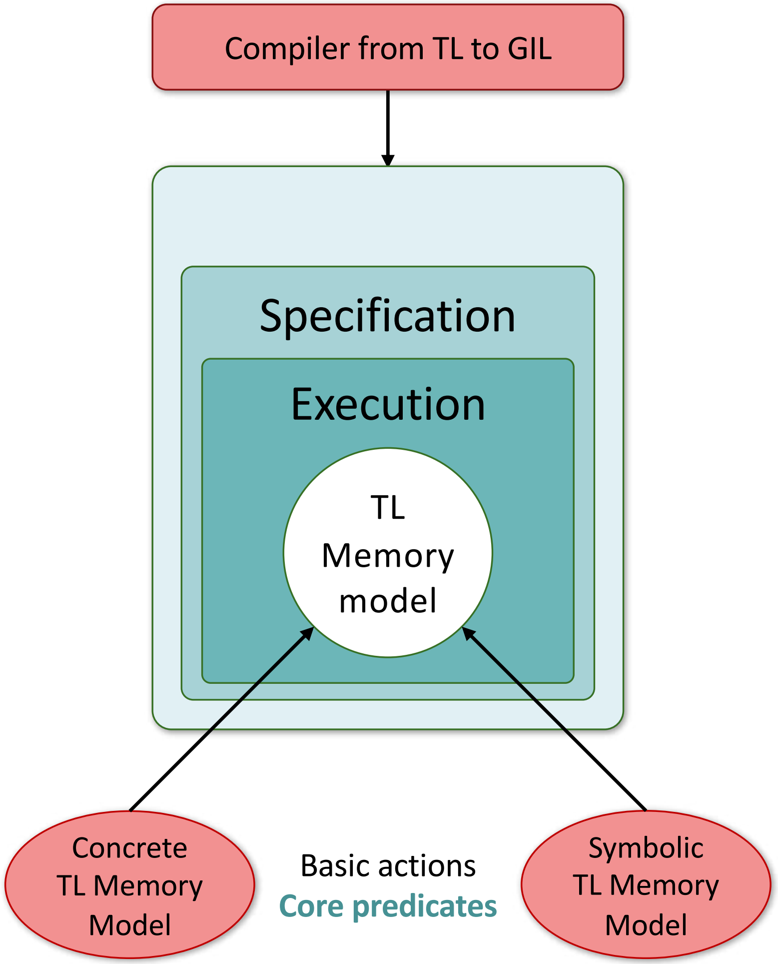
$$\{ P \} f(x) \{ Q \}$$

Specification

Step 1: **consume** the pre-condition

Using a unification algorithm, identify the part of the symbolic state that corresponds to the pre-condition and consume it, leaving the frame; in this process, we learn the bindings for the logical variables in the pre-condition

Aside: Specification Re-Use



Compiler from TL to GIL

Specification

Execution

TL Memory model

Concrete TL Memory Model

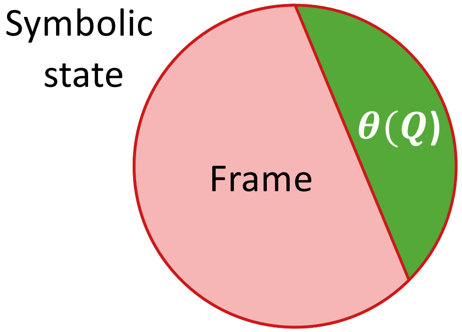
Basic actions

Core predicates

Symbolic TL Memory Model

Function specifications: $\{ P \} \, f(x) \, \{ Q \}$

Goal: apply a given function specification instead of symbolically executing a function



Symbolic state

Frame

$\theta(Q)$

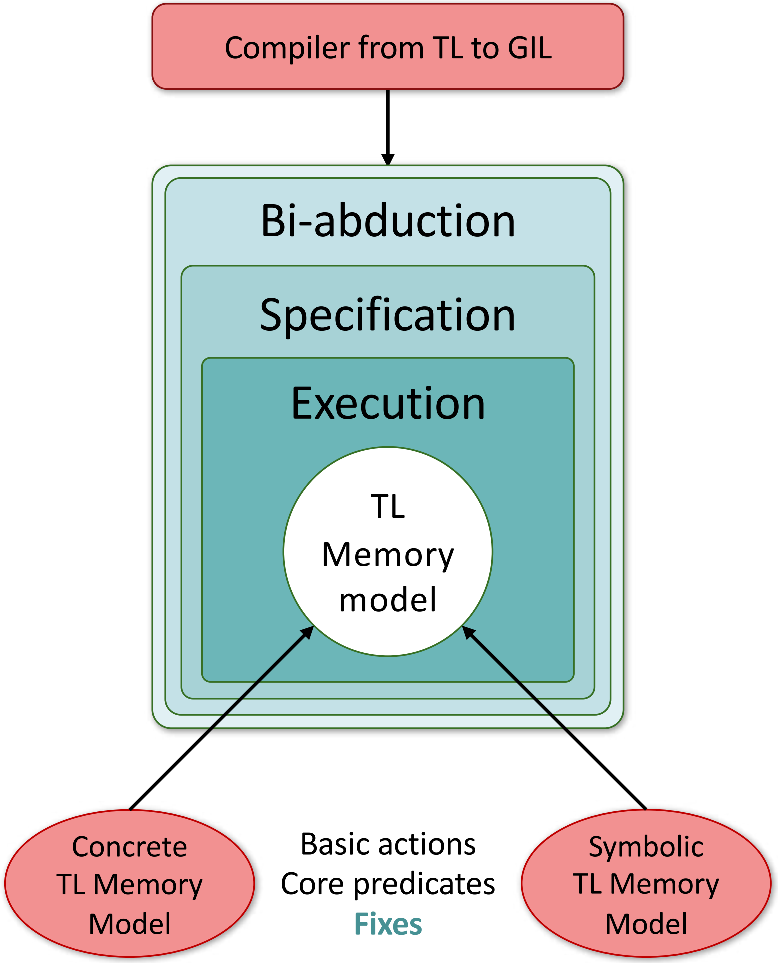
$\{ P \} \, f(x) \, \{ Q \}$

Specification

Step 1: **consume** the pre-condition

Step 2: **produce** the post-condition

Using the learned bindings, produce the resource corresponding to the post-condition



Compiler from TL to GIL

Bi-abduction

Specification

Execution

TL
Memory
model

Concrete
TL Memory
Model

Basic actions
Core predicates
Fixes

Symbolic
TL Memory
Model

Bi-abduction

Fundamental connection with execution engine **(POPL'19)**

User Input: Fixes

Missing information errors yield *fixes*, which represent ways of correcting the errors

An error can have multiple possible fixes that the tool developer needs to understand

Single additional rule

Bi-Action

$$\frac{\begin{array}{c}\mathsf{cmd}(\mathsf{p}, cs, i) = x := \alpha(e) \qquad \sigma.\mathrm{eval}_e(-) \rightsquigarrow (\sigma', \mathsf{v}) \\ \sigma'.\alpha(\mathsf{v}) \rightsquigarrow (-, [\mathcal{E}, \mathsf{v}']) \\ \mathrm{fix}(\mathsf{v}') \rightsquigarrow Q \qquad \sigma'.\mathrm{prod}(Q) \rightsquigarrow (\sigma'', -) \\ \sigma''.(\mathrm{setVar}_x \circ \alpha)(\mathsf{v}) \rightsquigarrow \sigma''' \end{array}}{\mathsf{p} \vdash \langle \sigma, cs, i \rangle \rightsquigarrow_a^{bi} \langle \sigma''', cs, i{+}1 \rangle}$$

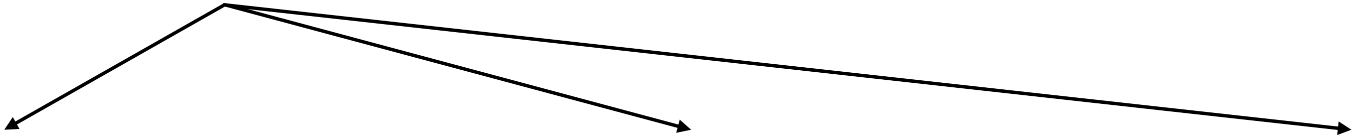
if an action fails with a given fix, produce that fix in the current state and re-execute the action

General Correctness Results

Parametric correctness results

Stated and proven independently of the underlying memory model

Minimal proof effort for the user



Execution

Correctness of symbolic actions w.r.t. concrete actions

Specification

Frame property for producers/consumers

Bi-abduction

After an action fix is applied, the execution can continue

General Correctness Results

Parametric correctness results

Symbolic Execution

- **Forward soundness** (analogy with Hoare triples)
- **Forward completeness** (no false positives)
- **Backward completeness** (no false positives, analogy with incorrectness triples)
- **Bounded verification guarantees**

ly of the underlying memory model

r

ecification

me property
lucers/consumers

Bi-abduction

After an action fix is applied,
the execution can continue

General Correctness Results

Parametric correctness results

Stated and proven independently of the underlying memory model

Minimal proof effort for the user

Execution

Correctness of symbolic
actions w.r.t. concrete actions

Specification

- **Full verification guarantees**

Bi-abduction

After an action fix is applied,
the execution can continue

General Correctness Results

Parametric correctness results

Stated and proven independently

Minimal proof effort for the user

Execution

Correctness of symbolic
actions w.r.t. concrete actions

Spec

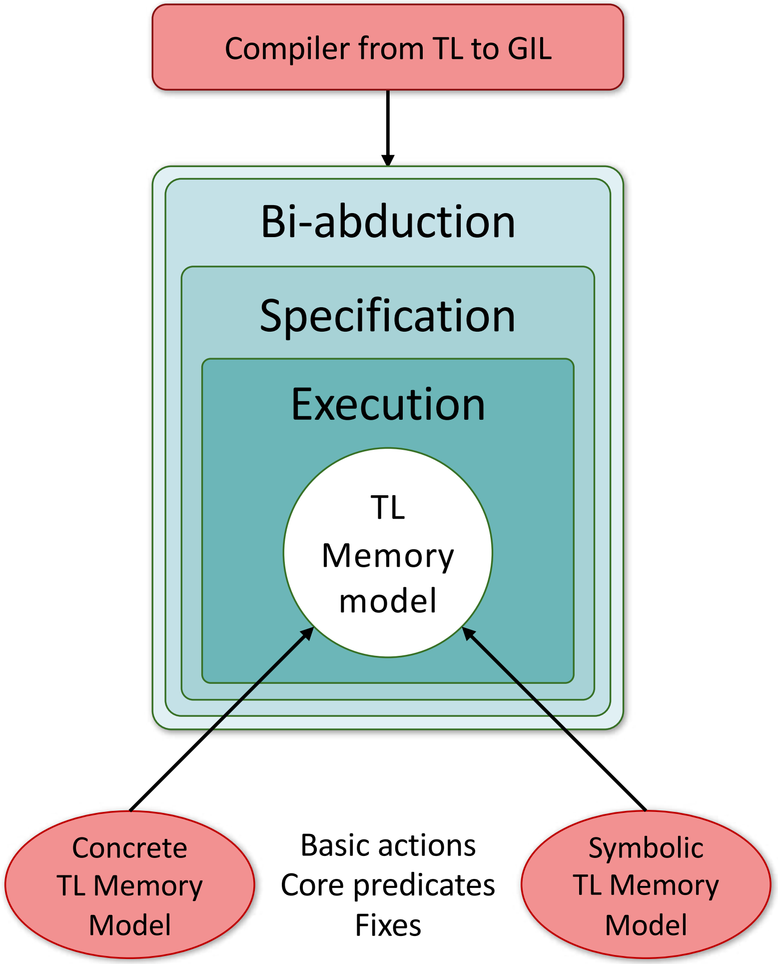
Frame
for produc

Bi-abduction

- **No false positives** if and only if
  - fixes are not over-approximating
  - no over-approximating specifications are used
- **Bounded verification guarantees** if and only if
  - fixes are not under-approximating

Gillian

INSTANTIATIONS



Gillian Instantiation

Compositional Memory Models

- TL **concrete** and **symbolic** memory models, using **basic actions**, **core predicates** and **fixes**
- The memory models are **compositional** to provide compositional analysis
- Basic actions must therefore account for positive, **negative** and missing information

Gillian-JS

- partial extensible object memory models
- explicit absence of object properties

(POPL'12, POPL'18, PPDP'18, POPL'19)

Gillian-C

- partial block-offset memory models
- explicit tracking of freed locations and block bounds

Gillian-JS

JS Compositional Memories

Concrete memory: $\mu \ : \ \mathcal{L} \rightharpoonup ((\mathcal{S} \rightharpoonup \mathcal{V}al_{\varnothing}) \times \wp(\mathcal{S})_{\bot} \times \mathcal{V}al_{\bot})$

$\varnothing$: absent

$\bot$: potentially missing

Gillian-JS

JS Compositional Memories

Concrete memory: $\mu \; : \; \mathcal{L} \rightharpoonup ((\mathcal{S} \rightharpoonup \mathcal{V}al_{\varnothing}) \times \wp(\mathcal{S})_{\perp} \times \mathcal{V}al_{\perp})$

location

$\varnothing$: absent

$\perp$: potentially missing

Gillian-JS

JS Compositional Memories

Concrete memory:

$$\mu \;:\; \underset{\substack{\uparrow \\ \text{location}}}{\mathcal{L}} \rightharpoonup ((\mathcal{S} \underset{\substack{\uparrow \\ \text{property table}}}{\rightharpoonup} \mathcal{V}al_{\varnothing}) \times \wp(\mathcal{S})_{\perp} \times \mathcal{V}al_{\perp})$$

$\varnothing$: absent

$\perp$: potentially missing

Gillian-JS

JS Compositional Memories

Concrete memory: $$\mu \ : \ \underset{\underset{\text{location}}{\uparrow}}{\mathcal{L}} \rightharpoonup ((\mathcal{S} \underset{\underset{\text{property table}}{\uparrow}}{\rightharpoonup} \mathcal{V}al_{\varnothing}) \times \underset{\underset{\textbf{domain table}}{\uparrow}}{\wp(\mathcal{S})_{\perp}} \times \mathcal{V}al_{\perp})$$

$\varnothing$: absent

$\perp$: potentially missing

Gillian-JS

JS Compositional Memories

Concrete memory: $$\mu \ : \ \underset{\underset{\text{location}}{\uparrow}}{\mathcal{L}} \rightharpoonup ((\mathcal{S} \underset{\underset{\text{property table}}{\uparrow}}{\rightharpoonup} \mathcal{V}al_{\varnothing}) \times \underset{\underset{\text{domain table}}{\uparrow}}{\wp(\mathcal{S})_{\bot}} \times \underset{\underset{\textbf{metadata}}{\uparrow}}{\mathcal{V}al_{\bot}})$$

$\varnothing$: absent

$\bot$: potentially missing

Gillian-JS

JS Compositional Memories

Concrete memory: $\mu \; : \; \mathcal{L} \rightharpoonup ((\mathcal{S} \rightharpoonup \mathcal{V}al_{\varnothing}) \times \wp(\mathcal{S})_{\perp} \times \mathcal{V}al_{\perp})$

Symbolic memory: $\hat{\mu} \; : \; \hat{\mathcal{E}}xpr \rightharpoonup ((\hat{\mathcal{E}}xpr \rightharpoonup \hat{\mathcal{E}}xpr_{\varnothing}) \times \hat{\mathcal{E}}xpr_{\perp} \times \hat{\mathcal{E}}xpr_{\perp})$

$\varnothing$: absent

$\perp$: potentially missing

Gillian-JS

JS Compositional Memories

Concrete memory: $\mu \ : \ \mathcal{L} \rightharpoonup ((\mathcal{S} \rightharpoonup \mathcal{V}al_{\varnothing}) \times \wp(\mathcal{S})_{\bot} \times \mathcal{V}al_{\bot})$

Symbolic memory: $\hat{\mu} \ : \ \hat{\mathcal{E}}xpr \rightharpoonup ((\hat{\mathcal{E}}xpr \rightharpoonup \hat{\mathcal{E}}xpr_{\varnothing}) \times \hat{\mathcal{E}}xpr_{\bot} \times \hat{\mathcal{E}}xpr_{\bot})$

$\varnothing$: absent

$\bot$: potentially missing

Well-formedness: Captures separation of object locations and properties within an object, as well as the connection between the domain table and the property table

Symbolic well-formedness:

$$\mathcal{W}f_{\pi}(\hat{\mu}) \triangleq \left(\pi \Rightarrow \bigwedge_{\substack{\hat{l},\hat{l}' \in \mathsf{dom}(\hat{\mu}) \\ \hat{l} \not\equiv \hat{l}'}} \hat{l} \neq \hat{l}' \ \wedge \bigwedge_{\substack{(\hat{h},-,-) \in \mathsf{codom}(\hat{\mu}) \\ \hat{p},\hat{p}' \in \mathsf{dom}(\hat{h}), \hat{p} \not\equiv \hat{p}'}} \hat{p} \neq \hat{p}' \ \wedge \bigwedge_{\substack{(\hat{h},\hat{d},-) \in \mathsf{codom}(\hat{\mu}) \\ \hat{d} \neq \bot}} \mathsf{dom}(\hat{h}) \subseteq \hat{d} \right)$$

Gillian-JS

JS Compositional Memories

Concrete memory: $\mu \ : \ \mathcal{L} \rightharpoonup ((\mathcal{S} \rightharpoonup \mathcal{V}al_{\varnothing}) \times \wp(\mathcal{S})_{\perp} \times \mathcal{V}al_{\perp})$

Symbolic memory: $\hat{\mu} \ : \ \hat{\mathcal{E}}xpr \rightharpoonup ((\hat{\mathcal{E}}xpr \rightharpoonup \hat{\mathcal{E}}xpr_{\varnothing}) \times \hat{\mathcal{E}}xpr_{\perp} \times \hat{\mathcal{E}}xpr_{\perp})$

$\varnothing$: absent

$\perp$: potentially missing

Basic actions, Core Predicates and Fixes

Six basic actions for the management of property table, domain table, and metadata

Three core predicates: $(\hat{l}, \hat{p}) \mapsto \hat{v}_{\emptyset}$, $\mathrm{domain}(\hat{l}, \hat{d})$, $\mathrm{metadata}(\hat{l}, \widehat{m})$

Exact fixes for all actions

Gillian-JS

JS Compositional Memories

Concrete memory: $\mu \;:\; \mathcal{L} \rightharpoonup ((\mathcal{S} \rightharpoonup \mathcal{V}al_{\varnothing}) \times \wp(\mathcal{S})_{\perp} \times \mathcal{V}al_{\perp})$

Symbolic memory: $\hat{\mu} \;:\; \hat{\mathcal{E}xpr} \rightharpoonup ((\hat{\mathcal{E}xpr} \rightharpoonup \hat{\mathcal{E}xpr}_{\varnothing}) \times \hat{\mathcal{E}xpr}_{\perp} \times \hat{\mathcal{E}xpr}_{\perp})$

$\varnothing$: absent

$\perp$: potentially missing

Basic actions, Core Predicates and Fixes

Six basic actions for the management of property table, domain table, and metadata

Three core predicates: $(\hat{l}, \hat{p}) \mapsto \hat{v}_{\emptyset}$, $\mathrm{domain}(\hat{l}, \hat{d})$, $\mathrm{metadata}(\hat{l}, \hat{m})$

Exact fixes for all actions

Explicit Negative Information: absence of object properties (expressed via core predicates)

$$(\hat{l}, \hat{p}) \mapsto \varnothing \qquad\qquad \text{domain}(\hat{l}, \hat{d}) \iff \forall \hat{p} \notin \hat{d}.\ (\hat{l}, \hat{p}) \mapsto \varnothing$$

Gillian-JS

JS Compositional Memories

Concrete memory: $\mu \; : \; \mathcal{L} \rightharpoonup ((\mathcal{S} \rightharpoonup \mathcal{V}al_{\varnothing}) \times \wp(\mathcal{S})_{\bot} \times \mathcal{V}al_{\bot})$

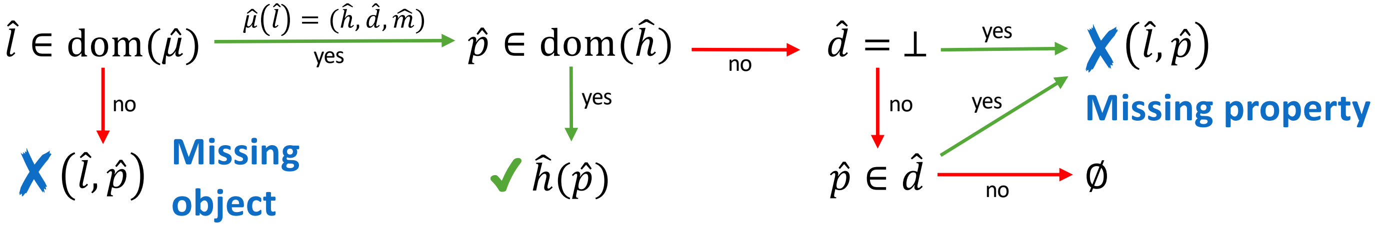
Symbolic memory: $\hat{\mu} \; : \; \hat{\mathcal{E}}xpr \rightharpoonup ((\hat{\mathcal{E}}xpr \rightharpoonup \hat{\mathcal{E}}xpr_{\varnothing}) \times \hat{\mathcal{E}}xpr_{\bot} \times \hat{\mathcal{E}}xpr_{\bot})$

$\varnothing$: absent

$\bot$: potentially missing

Actions account for positive, negative and missing information

Symbolic execution of action `getProp`$(\hat{l}, \hat{p})$:



Gillian-JS

JS Compositional Memories

Concrete memory: $\mu \ : \ \mathcal{L} \rightharpoonup ((\mathcal{S} \rightharpoonup \mathcal{V}al_{\varnothing}) \times \wp(\mathcal{S})_{\bot} \times \mathcal{V}al_{\bot})$

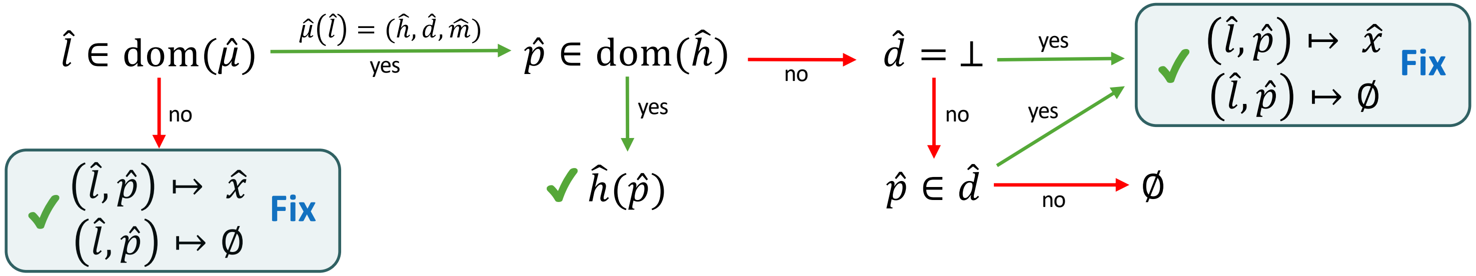
Symbolic memory: $\hat{\mu} \ : \ \hat{\mathcal{E}}xpr \rightharpoonup ((\hat{\mathcal{E}}xpr \rightharpoonup \hat{\mathcal{E}}xpr_{\varnothing}) \times \hat{\mathcal{E}}xpr_{\bot} \times \hat{\mathcal{E}}xpr_{\bot})$

$\varnothing$: absent

$\bot$: potentially missing

Actions account for positive, negative and missing information

Bi-abductive execution of action `getProp`$(\hat{l}, \hat{p})$:



$\hat{l} \in \mathrm{dom}(\hat{\mu}) \xrightarrow[\text{yes}]{\hat{\mu}(\hat{l}) = (\hat{h}, \hat{d}, \hat{m})} \hat{p} \in \mathrm{dom}(\hat{h}) \xrightarrow[\text{no}]{} \hat{d} = \bot \xrightarrow{\text{yes}}$ ✓ $(\hat{l}, \hat{p}) \mapsto \hat{x}$, $(\hat{l}, \hat{p}) \mapsto \emptyset$ **Fix**

$\hat{l} \in \mathrm{dom}(\hat{\mu}) \xrightarrow{\text{no}}$ ✓ $(\hat{l}, \hat{p}) \mapsto \hat{x}$, $(\hat{l}, \hat{p}) \mapsto \emptyset$ **Fix**

$\hat{p} \in \mathrm{dom}(\hat{h}) \xrightarrow{\text{yes}}$ ✓ $\hat{h}(\hat{p})$

$\hat{d} = \bot \xrightarrow{\text{no}} \hat{p} \in \hat{d}$

$\hat{p} \in \hat{d} \xrightarrow{\text{yes}}$ ✓ $(\hat{l}, \hat{p}) \mapsto \hat{x}$, $(\hat{l}, \hat{p}) \mapsto \emptyset$ **Fix**

$\hat{p} \in \hat{d} \xrightarrow[\text{no}]{} \emptyset$

Gillian-C

C Simplified Compositional Memories

Concrete memory: $\mu \; : \; \mathcal{L} \rightharpoonup ((\mathbb{N} \rightharpoonup \mathcal{V}al) \times \mathbb{N}_{\perp})_{\varnothing}$

$\varnothing$: freed

$\perp$: potentially missing

Gillian-C

C Simplified Compositional Memories

Concrete memory: $\mu \; : \; \underset{\overset{\uparrow}{\text{location}}}{\mathcal{L}} \rightharpoonup ((\mathbb{N} \rightharpoonup \mathcal{V}al) \times \mathbb{N}_{\bot})_{\varnothing}$

$\varnothing$: freed

$\bot$: potentially missing

Gillian-C

C Simplified Compositional Memories

Concrete memory: $\mu \;:\; \mathcal{L} \rightharpoonup ((\mathbb{N} \rightharpoonup \mathcal{V}al) \times \mathbb{N}_{\perp})_{\varnothing}$

location

block contents

$\varnothing$: freed

$\perp$: potentially missing

Gillian-C

C Simplified Compositional Memories

Concrete memory: $\mu \; : \; \underset{\substack{\uparrow \\ \text{location}}}{\mathcal{L}} \rightharpoonup ((\underset{\substack{\uparrow \\ \text{block contents}}}{\mathbb{N} \rightharpoonup \mathcal{V}al}) \times \underset{\substack{\uparrow \\ \textbf{block bound}}}{\mathbb{N}_\perp})_\varnothing$

$\varnothing$: freed

$\perp$: potentially missing

Gillian-C

C Simplified Compositional Memories

Concrete memory: $\mu \;:\; \mathcal{L} \rightharpoonup ((\mathbb{N} \rightharpoonup \mathcal{V}al) \times \mathbb{N}_{\perp})_{\varnothing}$

Symbolic memory: $\hat{\mu} \;:\; \hat{\mathcal{E}}xpr \rightharpoonup ((\hat{\mathcal{E}}xpr \rightharpoonup \hat{\mathcal{E}}xpr) \times \hat{\mathcal{E}}xpr_{\perp})_{\varnothing}$

$\varnothing$: freed

$\perp$: potentially missing

Gillian-C

C Simplified Compositional Memories

Concrete memory: $\mu \ : \ \mathcal{L} \rightharpoonup ((\mathbb{N} \rightharpoonup \mathcal{V}al) \times \mathbb{N}_{\perp})_{\varnothing}$

Symbolic memory: $\hat{\mu} \ : \ \hat{\mathcal{E}}xpr \rightharpoonup ((\hat{\mathcal{E}}xpr \rightharpoonup \hat{\mathcal{E}}xpr) \times \hat{\mathcal{E}}xpr_{\perp})_{\varnothing}$

$\varnothing$: freed

$\perp$: potentially missing

Well-formedness: Captures separation of block locations and offsets within a block, as well as the connection between the block bound and the block contents

Symbolic well-formedness:

$$\mathcal{W}f_{\pi}(\hat{\mu}) \triangleq \left(\pi \Rightarrow \bigwedge_{\substack{\hat{l},\hat{l}' \in \mathsf{dom}(\hat{\mu}) \\ \hat{l} \neq \hat{l}'}} \hat{l} \neq \hat{l}' \wedge \bigwedge_{\substack{(\hat{k},-) \in \mathsf{codom}(\hat{\mu}) \\ \hat{o},\hat{o}' \in \mathsf{dom}(\hat{k}),\, \hat{o} \neq \hat{o}'}} \hat{o} \neq \hat{o}' \wedge \bigwedge_{\substack{(\hat{k},\hat{n}) \in \mathsf{codom}(\hat{\mu}) \\ \hat{n} \neq \perp,\, \hat{o} \in \mathsf{dom}(\hat{k})}} \hat{o} < \hat{n} \right)$$

Gillian-C

C Simplified Compositional Memories

Concrete memory: $\mu \; : \; \mathcal{L} \rightharpoonup ((\mathbb{N} \rightharpoonup \mathcal{V}al) \times \mathbb{N}_{\perp})_{\varnothing}$

Symbolic memory: $\hat{\mu} \; : \; \hat{\mathcal{E}}xpr \rightharpoonup ((\hat{\mathcal{E}}xpr \rightharpoonup \hat{\mathcal{E}}xpr) \times \hat{\mathcal{E}}xpr_{\perp})_{\varnothing}$

$\varnothing$: freed

$\perp$: potentially missing

Basic actions, Core Predicates and Fixes

Six basic actions for the management of blocks, bounds, and freed objects

Three core predicates: $(\hat{l}, \hat{o}) \mapsto \hat{v}$, $\mathrm{bound}(\hat{l}, \hat{n})$, $\hat{l} \mapsto \emptyset$

Exact fixes for all actions

Gillian-C

C Simplified Compositional Memories

Concrete memory: $\mu \ : \ \mathcal{L} \rightharpoonup ((\mathbb{N} \rightharpoonup \mathcal{V}al) \times \mathbb{N}_{\perp})_{\varnothing}$

Symbolic memory: $\hat{\mu} \ : \ \hat{\mathcal{E}}xpr \rightharpoonup ((\hat{\mathcal{E}}xpr \rightharpoonup \hat{\mathcal{E}}xpr) \times \hat{\mathcal{E}}xpr_{\perp})_{\varnothing}$

$\varnothing$: freed

$\perp$: potentially missing

Basic actions, Core Predicates and Fixes

Six basic actions for the management of blocks, bounds, and freed objects

Three core predicates: $(\hat{l}, \hat{o}) \mapsto \hat{v}$, $\mathrm{bound}(\hat{l}, \hat{n})$, $\hat{l} \mapsto \emptyset$

Exact fixes for all actions

Explicit Negative Information: freed locations and block bounds (expressed via core predicates)

$$\hat{l} \mapsto \varnothing \qquad\qquad \mathrm{bound}(\hat{l}, \hat{n})$$

Gillian-C

C Simplified Compositional Memories

Concrete memory: $\mu \ : \ \mathcal{L} \rightharpoonup ((\mathbb{N} \rightharpoonup \mathcal{V}al) \times \mathbb{N}_{\perp})_{\varnothing}$

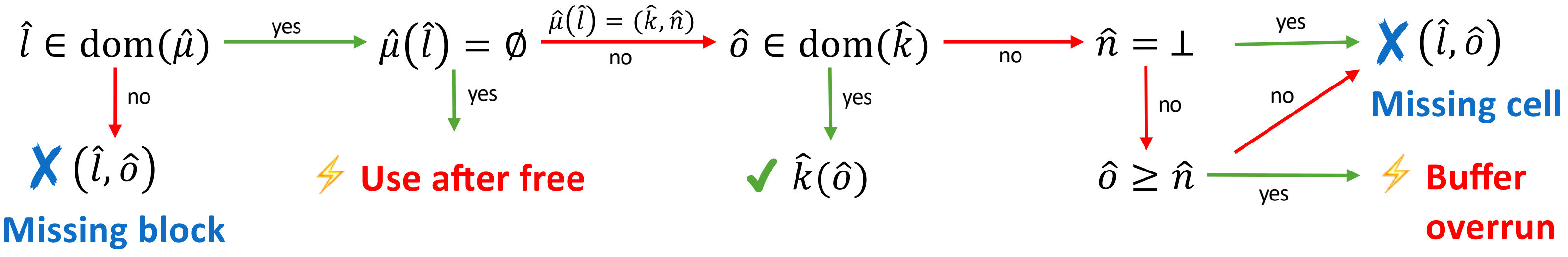
Symbolic memory: $\hat{\mu} \ : \ \hat{\mathcal{E}}xpr \rightharpoonup ((\hat{\mathcal{E}}xpr \rightharpoonup \hat{\mathcal{E}}xpr) \times \hat{\mathcal{E}}xpr_{\perp})_{\varnothing}$

$\varnothing$: freed

$\perp$: potentially missing

Actions account for positive, negative and missing information

Symbolic execution of the action `getCell`$(\hat{l}, \hat{o})$:



$\hat{l} \in \mathrm{dom}(\hat{\mu})$

yes

$\hat{\mu}(\hat{l}) = \emptyset$

$\hat{\mu}(\hat{l}) = (\hat{k}, \hat{n})$

no

$\hat{o} \in \mathrm{dom}(\hat{k})$

no

$\hat{n} = \perp$

yes

✗ $(\hat{l}, \hat{o})$

Missing cell

no

✗ $(\hat{l}, \hat{o})$

Missing block

yes

⚡ **Use after free**

yes

✓ $\hat{k}(\hat{o})$

no

no

$\hat{o} \geq \hat{n}$

yes

⚡ **Buffer overrun**

Gillian-C

C Simplified Compositional Memories

Concrete memory: $\mu \;:\; \mathcal{L} \rightharpoonup ((\mathbb{N} \rightharpoonup \mathcal{V}al) \times \mathbb{N}_{\perp})_{\varnothing}$

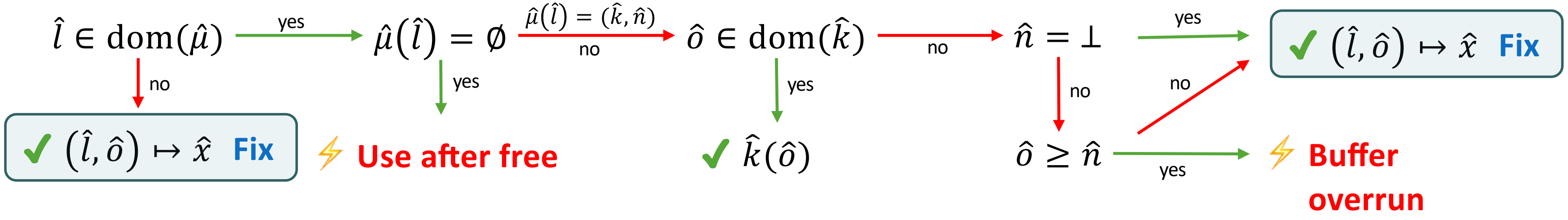
Symbolic memory: $\hat{\mu} \;:\; \hat{\mathcal{E}}xpr \rightharpoonup ((\hat{\mathcal{E}}xpr \rightharpoonup \hat{\mathcal{E}}xpr) \times \hat{\mathcal{E}}xpr_{\perp})_{\varnothing}$

$\varnothing$: freed

$\perp$: potentially missing

Actions account for positive, negative and missing information

Bi-abductive execution of the action `getCell`$(\hat{l}, \hat{o})$:



$\hat{l} \in \mathrm{dom}(\hat{\mu})$

yes

no

✓ $(\hat{l}, \hat{o}) \mapsto \hat{x}$ **Fix**

$\hat{\mu}(\hat{l}) = \emptyset$

yes

⚡ **Use after free**

$\hat{\mu}(\hat{l}) = (\hat{k}, \hat{n})$

no

$\hat{o} \in \mathrm{dom}(\hat{k})$

yes

✓ $\hat{k}(\hat{o})$

no

$\hat{n} = \perp$

yes

✓ $(\hat{l}, \hat{o}) \mapsto \hat{x}$ **Fix**

no

$\hat{o} \geq \hat{n}$

no

yes

⚡ **Buffer overrun**

Gillian

IN PRACTICE

Symbolic Testing: Buckets.js and Collections-C

Stand-alone real-world data-structure libraries for JavaScript and C

Buckets.js: ~1.5Kloc, > 65K downloads on npm

Data Structure	Symbolic Tests	Executed GIL cmds	Time
array	9	329,854	2.53s
bag	7	1,343,808	4.78s
bst	11	3,750,552	12.47s
dict	7	401,964	1.81s
heap	4	1,487,554	3.36s
llist	9	588,699	3.97s
mdict	6	1,106,058	3.84s
queue	6	407,061	2.22s
pqueue	5	2,297,943	4.02s
set	6	2,181,474	4.56s
stack	4	306,434	1.63s
Total	**73**	**14,201,401**	**45.19s**

100% line coverage, 3 bugs found and fixed

Collections-C: ~5.5Kloc, 2K stars on GitHub

Data Structure	Symbolic Tests	Executed GIL Cmds	Time
array	22	109,290	4.21s
deque	34	106,737	6.57s
list	37	730,655	13.02s
pqueue	4	39,828	0.65s
queue	2	15,726	0.64s
pqueue	3	27,284	0.52s
queue	38	325,383	7.18s
stack	2	5,211	0.28s
treetbl	13	618,326	2.98s
treeset	6	108,583	3.29s
Total	**161**	**2,097,023**	**39.34s**

Bugs found in library and concrete tests, fixed

Symbolic Testing: Cash Events Module (ECOOP'20)

Cash: A compact alternative for jQuery, > 450K downloads on npm, 4.4K stars on GitHub
Uses DOM Core Level 1, DOM UI Events, JS promises, await/async

Gabriela Sampaio

8 symbolic tests, 100% line coverage

Test Name	rHand	sHand	tOff	other	Total
Time (s)	5.54	144.38	22.87	66.53	**239.34**
Executed GIL cmds	1,468,907	38,240,506	9,400,471	23,439,230	**72,549,114**

Bounded Correctness Guarantees

rHand: If a handler has been triggered, then it must have previously been registered

sHand: If a single handler has been registered to a given event, then that is the only handler that can be triggered for that event **(revealed two bugs, fixed)**

Correctness bound: length of the event type is at most 20 characters

Full Verification: AWS Encryption SDK

Target code: AWS Encryption SDK message header manipulation in JS and C

Current approach to validation:

JS: concrete testing, runtime correctness assertions
C: concrete testing, runtime correctness assertions, bounded model checking (CBMC)

```
/* Precondition: readPos must be non-negative and within the byte length of the buffer given. */
needs(
  readPos >= 0 && dataView.byteLength >= readPos,
  'readPos out of bounds.'
)

/* Precondition: elementCount and fieldsPerElement must be non-negative. */
needs(
  elementCount >= 0 && fieldsPerElement >= 0,
  'elementCount and fieldsPerElement must be non-negative.'
)
```

```
int aws_cryptosdk_enc_ctx_deserialize(
    struct aws_allocator *alloc, struct aws_hash_table *enc_ctx, struct aws_byte_cursor *cursor) {
    AWS_PRECONDITION(aws_allocator_is_valid(alloc));
    AWS_PRECONDITION(aws_hash_table_is_valid(enc_ctx));
    AWS_PRECONDITION(aws_byte_cursor_is_valid(cursor));

    aws_cryptosdk_enc_ctx_clear(enc_ctx);

    if (cursor->len == 0) {
        AWS_POSTCONDITION(aws_allocator_is_valid(alloc));
        AWS_POSTCONDITION(aws_hash_table_is_valid(enc_ctx));
        AWS_POSTCONDITION(aws_byte_cursor_is_valid(cursor));
        return AWS_OP_SUCCESS;
    }
```

First project: verification of the message header deserialisation module

(~200loc for JS, ~950loc for C, using full features of both languages)

AWS Verification: Header Deserialisation

Results: Gillian-JS and Gillian-C verify that the JS and C deserialisation modules:

- correctly deserialise a well-formed header
- return false (JS) or throw an appropriate error (C) if supplied an incomplete header
- throw an appropriate error if supplied a malformed header

Impact on AWS code and Gillian:

- improved the implementation of the JS readElements auxiliary function
- discovered **one bug** and **one vulnerability** in the JS decodeEncryptionContext function
- found one **over-allocation** and one **undefined behaviour** in the aws-c-common library
- substantially improved the reasoning capabilities of Gillian

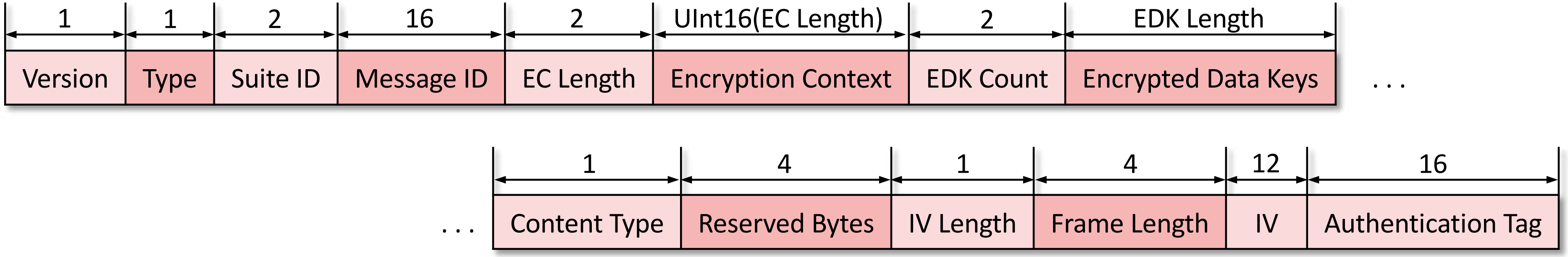
Workload:

- ~2 person months for JS, ~1 person-month for C
- ~3.5K lines of annotations (predicates, specifications, invariants, lemmas, proof tactics)

(~1.2K language-independent, ~1K for C, ~1.3K for JS)

AWS SDK Message Header

A sequence of bytes, divided into sections



1	1	2	16	2	UInt16(EC Length)	2	EDK Length	
Version	Type	Suite ID	Message ID	EC Length	Encryption Context	EDK Count	Encrypted Data Keys	. . .

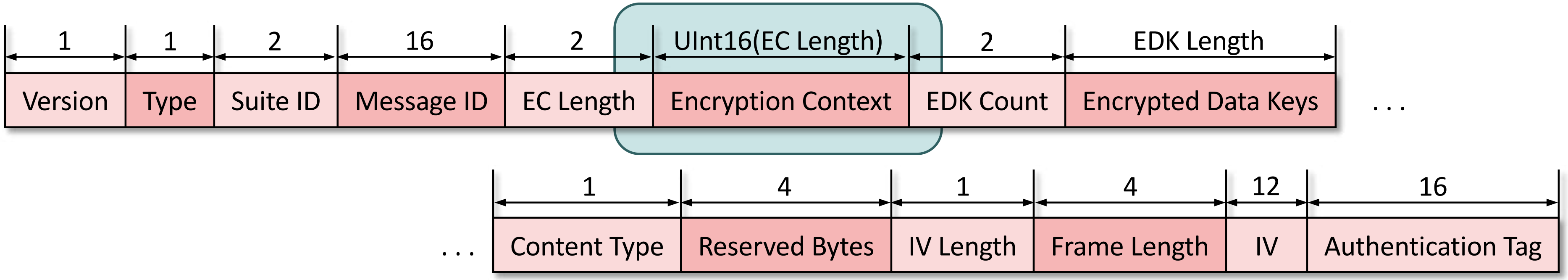
	1	4	1	4	12	16
. . .	Content Type	Reserved Bytes	IV Length	Frame Length	IV	Authentication Tag

Our Approach:

- build **language-independent** first-order abstractions capturing the header structure
- using these abstractions, build **language-specific** abstractions capturing header-related objects and structures in JS and C memories used in the AWS SDK implementations
- prove lemmas about all abstractions
- specify and verify all functions of the deserialisation modules

AWS SDK Message Header

A sequence of bytes, divided into sections



1	1	2	16	2	UInt16(EC Length)	2	EDK Length	
Version	Type	Suite ID	Message ID	EC Length	Encryption Context	EDK Count	Encrypted Data Keys	. . .

	1	4	1	4	12	16
. . .	Content Type	Reserved Bytes	IV Length	Frame Length	IV	Authentication Tag

Our Approach:

- build **language-independent** first-order abstractions capturing the header structure
- using these abstractions, build **language-specific** abstractions capturing header-related objects

FOCUS: Encryption context & related functions

- Section of variable length, complex to specify and reason about
- Source of the JS bugs

AWS: LANGUAGE-INDEPENDENT SPECIFICATION

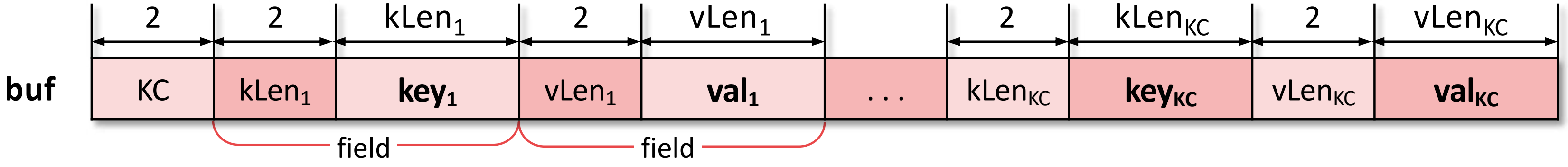
Specification: Encryption Context

Encryption context: serialised list of key-value pairs

	2	2	$kLen_1$	2	$vLen_1$		2	$kLen_{KC}$	2	$vLen_{KC}$
buf	KC	$kLen_1$	$\mathbf{key_1}$	$vLen_1$	$\mathbf{val_1}$	. . .	$kLen_{KC}$	$\mathbf{key_{KC}}$	$vLen_{KC}$	$\mathbf{val_{KC}}$

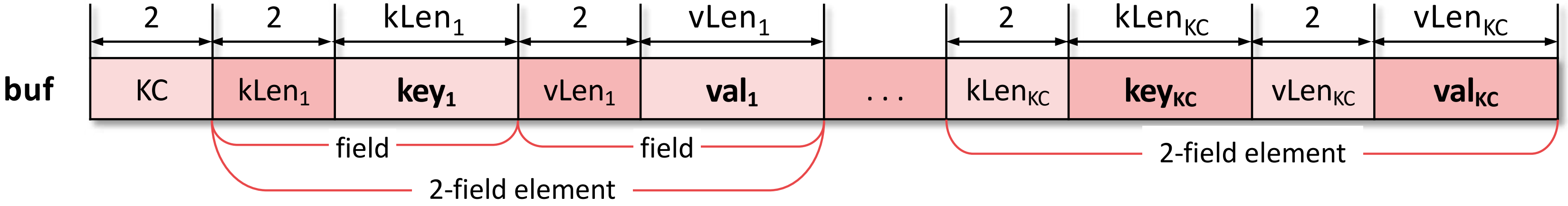
Specification: Encryption Context

Encryption context: serialised list of key-value pairs



Specification: Encryption Context

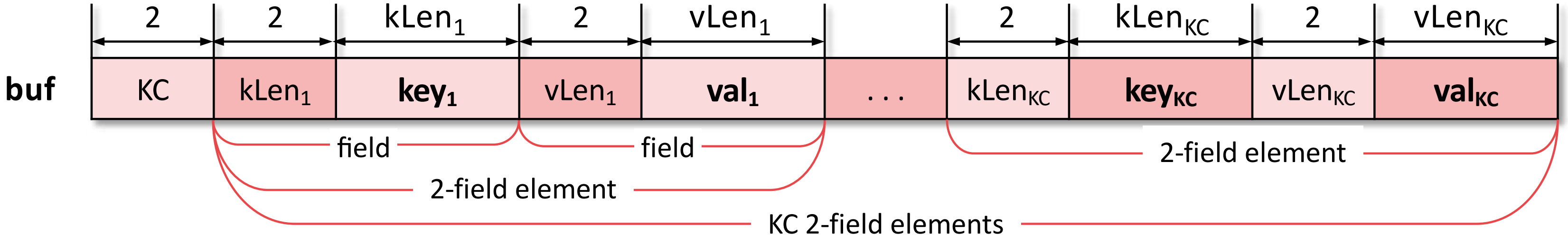
Encryption context: serialised list of key-value pairs



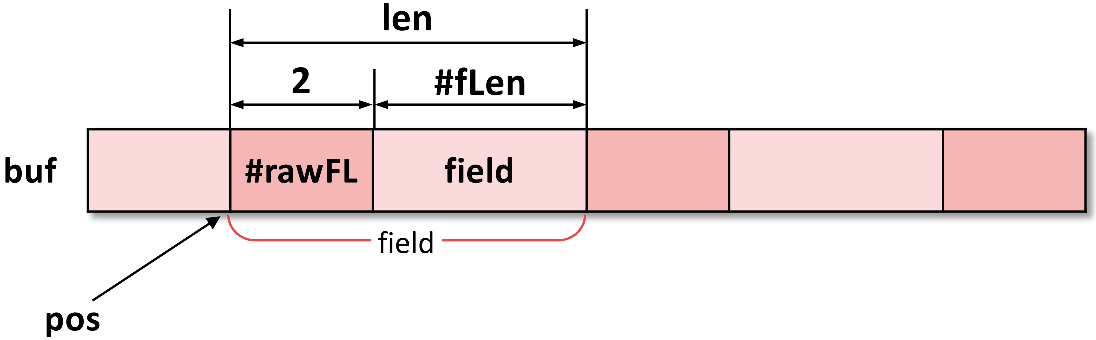
	2	2	$kLen_1$	2	$vLen_1$		2	$kLen_{KC}$	2	$vLen_{KC}$
buf	KC	$kLen_1$	**key_1**	$vLen_1$	**val_1**	. . .	$kLen_{KC}$	**key_{KC}**	$vLen_{KC}$	**val_{KC}**

Specification: Encryption Context

Encryption context: serialised list of key-value pairs



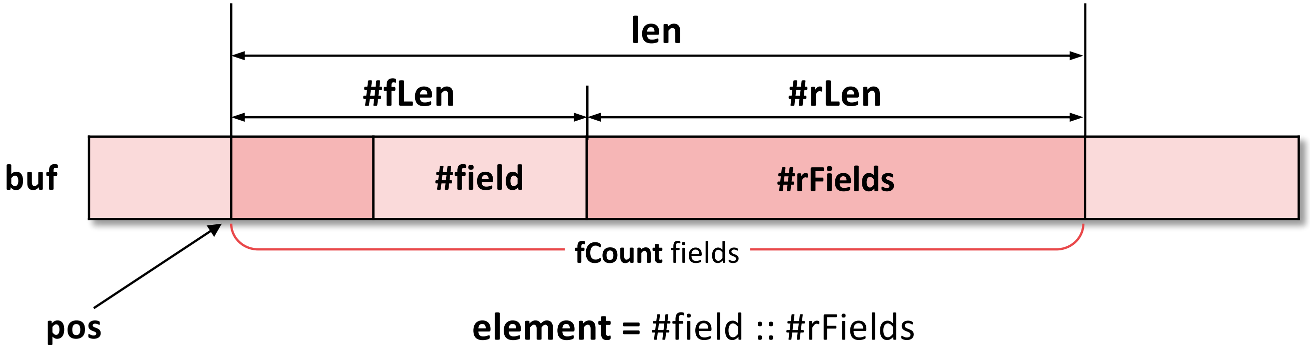
Specification: Field



```
pred Field(buf : byte list, pos : int, field : byte list, len : int)

  (0 <= pos) *
  (#rawFL = l-sub(buf, pos, 2)) *          // l-sub(buf, pos, n): sublist of buf at pos of length n
  UInt16(#rawFL, #fLen) *                  // Conversion to an unsigned 16-bit integer
  (field = l-sub(buf, pos + 2, #fLen)) *
  (len = 2 + #fLen) *
  (pos + len <= l-len buf);                // Field must fit in buffer
```

Abstractions: language-independent, pure

Specification: Complete Element

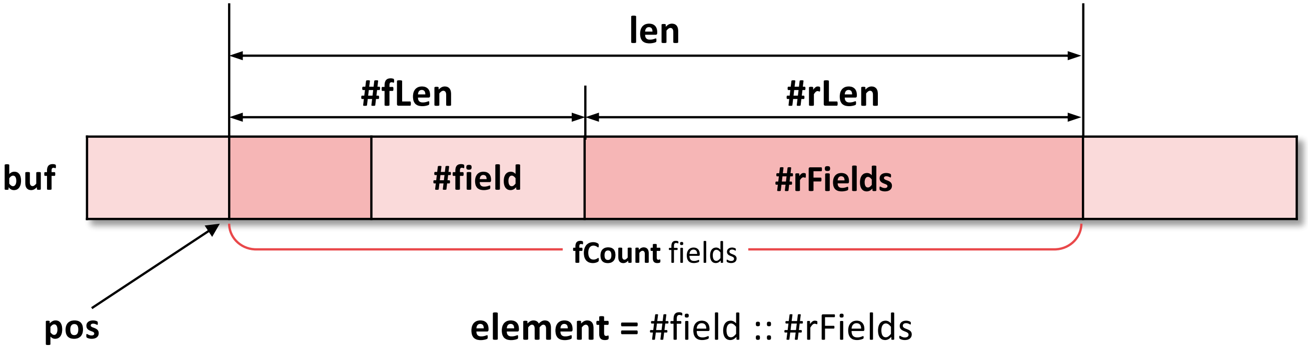


element = #field :: #rFields

```
pred CElement(buf : byte list, pos : int, fCount : int, element : (byte list) list, len : int) :

  (0 <= pos) ∗ (pos <= l-len buf) ∗                          // Base case: no more fields to read
  (fCount = 0) ∗ (element = [ ]) ∗ (len = 0),

  (0 < fCount) ∗                                             // Inductive case: first field and rest
  Field(buf, pos, #field, #fLen) ∗
  CElement(buf, pos + #fLen, fCount − 1, #rFields, #rLen) ∗
  (element = #field :: #rFields) ∗
  (len = #fLen + #rLen);
```

Abstractions: language-independent, pure

Specification: Complete Element



element = #field :: #rFields

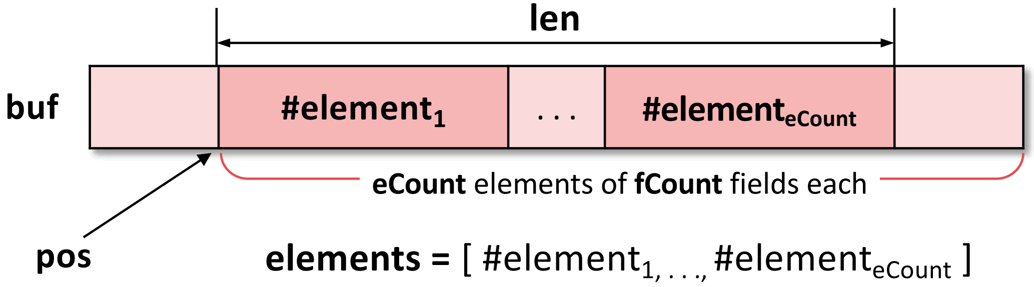
```
pred CElement(buf : byte list, pos : int, fCount : int, element : (byte list) list, len : int) :
```

Additionally:

- **Incomplete element**: part of an element with correct structure
- **Broken element**: element with incorrect structure
- A general **Element** abstraction incorporating all three types of elements

Abstractions: language-independent, pure

Specification: Complete Element Sequence



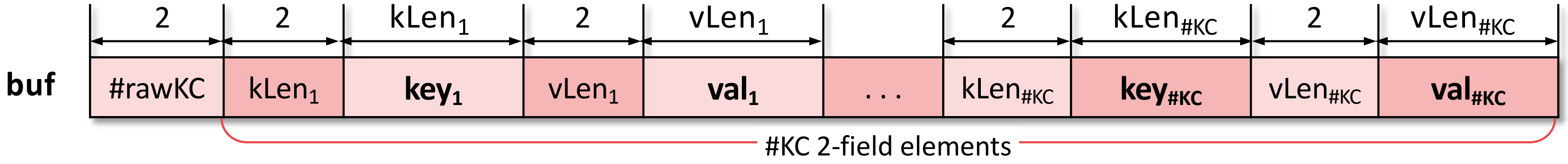
elements = [#element$_{1, \dots,}$ #element$_{eCount}$]

```
pred CElements(buf : byte list, pos : int, eCount : int, fCount : int,
               elements : ((byte list) list) list, len : int)
// The buffer buf contains, at position pos, a sequence of eCount complete elements,
// each consisting of fCount fields, with overall contents denoted by elements (list of
// lists of field contents) and total length len
```

Abstractions: language-independent, pure

Specification: Complete Encryption Context

Encryption context: serialised list of key-value pairs



	2	2	kLen$_1$	2	vLen$_1$		2	kLen$_{\#KC}$	2	vLen$_{\#KC}$
buf	#rawKC	kLen$_1$	**key$_1$**	vLen$_1$	**val$_1$**	. . .	kLen$_{\#KC}$	**key$_{\#KC}$**	vLen$_{\#KC}$	**val$_{\#KC}$**

#KC 2-field elements

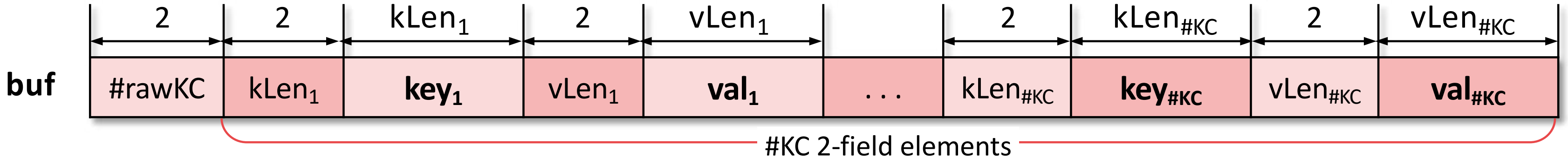
KVs = [[key$_1$, val$_1$], …, [key$_{\#KC}$, val$_{\#KC}$]]

```
pred CEncryptionContext(buf  : byte list, KVs : ((byte list) list) list) :
  (buf = [ ]) ∗ (KVs = [ ]),                        // The EC is empty, no key−value pairs

  (#rawKC = l−sub(buf, 0, 2)) ∗                     // EC not empty: learn raw key count
  UInt16(#rawKC, #KC) ∗ (0 < #KC) ∗                 // Learn actual key count, which must be positive
  CElements(buf, 2, #KC, 2, KVs, #ECLen) ∗          // #KC elements, each with 2 fields (key−value pairs)
  FirstProj(KVs, #keys) ∗ Unique(#keys) ∗           // The keys (first projection) must be unique
  (2 + #ECLen = l−len buf);                         // And the EC must fill the buffer completely
```

Abstractions: language-independent, pure

Specification: Complete Encryption Context

Encryption context: serialised list of key-value pairs



	2	2	$kLen_1$	2	$vLen_1$		2	$kLen_{\#KC}$	2	$vLen_{\#KC}$
buf	#rawKC	$kLen_1$	$\mathbf{key_1}$	$vLen_1$	$\mathbf{val_1}$	. . .	$kLen_{\#KC}$	$\mathbf{key_{\#KC}}$	$vLen_{\#KC}$	$\mathbf{val_{\#KC}}$

#KC 2-field elements

KVs = [[key_1, val_1], …, [$key_{\#KC}$, $val_{\#KC}$]]

```
pred CEncryptionContext(buf  : byte list, KVs : ((byte list) list) list) :
```

Analogously to elements:

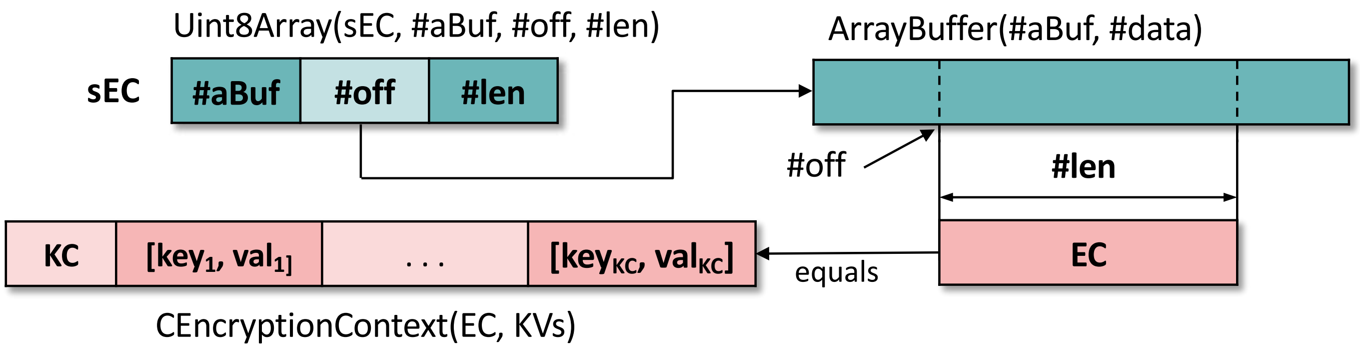
- Additional abstractions capturing **incomplete** and **broken** encryption contexts
- A general **EncryptionContext** abstraction incorporating all three types of elements

AWS: Language-dependent specification for JS and C

JS: Serialised Encryption Context

In JS, the serialised encryption context is accessible via an ES6 UInt8Array object

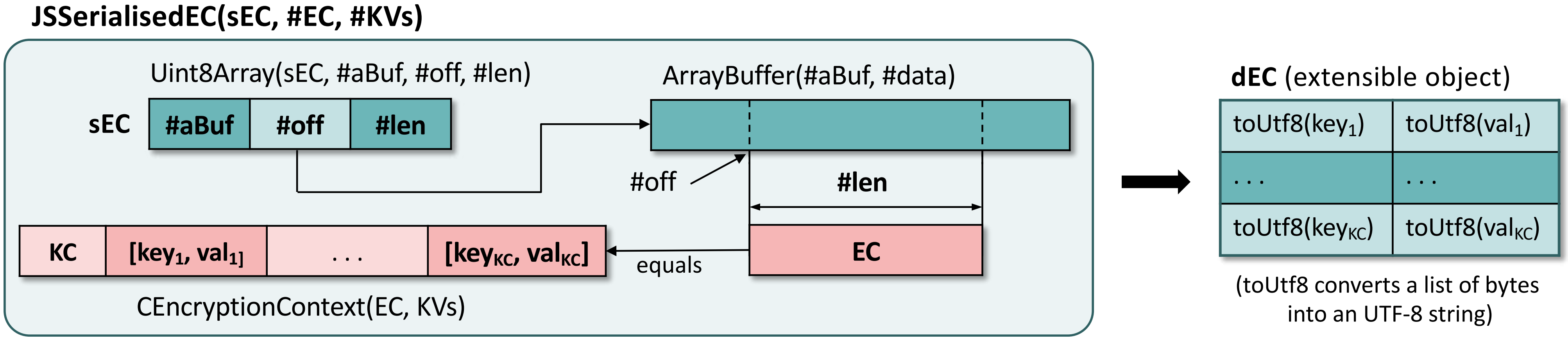
```
pred JSSerialisedEC(sEC : Obj, EC : byte list, KVs : ((byte list) list) list) :
  Uint8Array(sEC, #aBuf, #off, #len) ∗  // sEC is a UInt8Array on top of the ArrayBuffer #aBuf
  ArrayBuffer(#aBuf, #data) ∗          // which holds the information #data (a list of bytes),
  (EC == l-sub(#aBuf, #off, #len) ∗     // of which the encryption context EC is part,
  CEncryptionContext(EC, KVs)          // and the EC contains the key−value pairs KVs
```



Abstractions: language-dependent, resource; language-independent, pure

JS: Deserialised Encryption Context

In JS, the encryption context is deserialised into a JS object representing a key-value map



JSSerialisedEC(sEC, #EC, #KVs)

Uint8Array(sEC, #aBuf, #off, #len)

ArrayBuffer(#aBuf, #data)

sEC | **#aBuf** | **#off** | **#len**

#off

#len

KC | **[key$_1$, val$_1$]** | . . . | **[key$_{KC}$, val$_{KC}$]**

equals

EC

CEncryptionContext(EC, KVs)

dEC (extensible object)

toUtf8(key$_1$)	toUtf8(val$_1$)
. . .	. . .
toUtf8(key$_{KC}$)	toUtf8(val$_{KC}$)

(toUtf8 converts a list of bytes into an UTF-8 string)

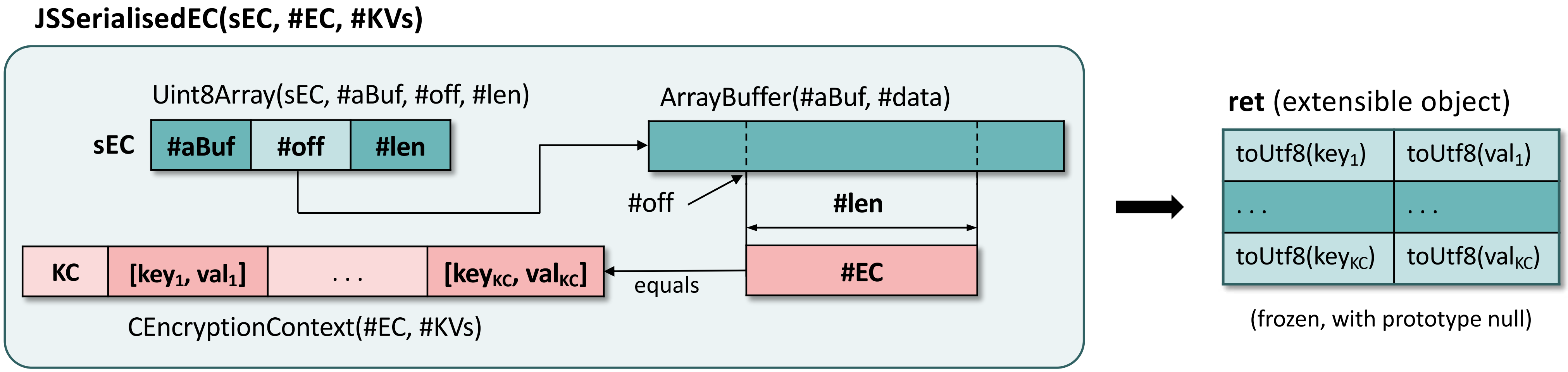
- The keys, when encoded with toUtf8, must be unique
- The resulting key-value map should be frozen to prevent tampering

```
pred JSDeserialisedEC(dEC : Obj, KVs : ((byte list) list) list) :
  FrozenObject(dEC, null, #utf8KVs) *  // dEC is a frozen JS object with prototype null
                                       // that holds the property-value pairs given by #utf8KVs
  toUtf8(KVs, #utf8KVs)                // which are obtained from KVs by converting to UTF-8
```

Abstractions: language-dependent, resource; language-independent, pure

JS Specification: Header Deserialisation

decodeEncryptionContext: deserialises the encryption context in JS



JSSerialisedEC(sEC, #EC, #KVs)

Uint8Array(sEC, #aBuf, #off, #len)

sEC **#aBuf** **#off** **#len**

ArrayBuffer(#aBuf, #data)

#off

#len

KC **[key$_1$, val$_1$]** . . . **[key$_{KC}$, val$_{KC}$]**

equals

#EC

CEncryptionContext(#EC, #KVs)

ret (extensible object)

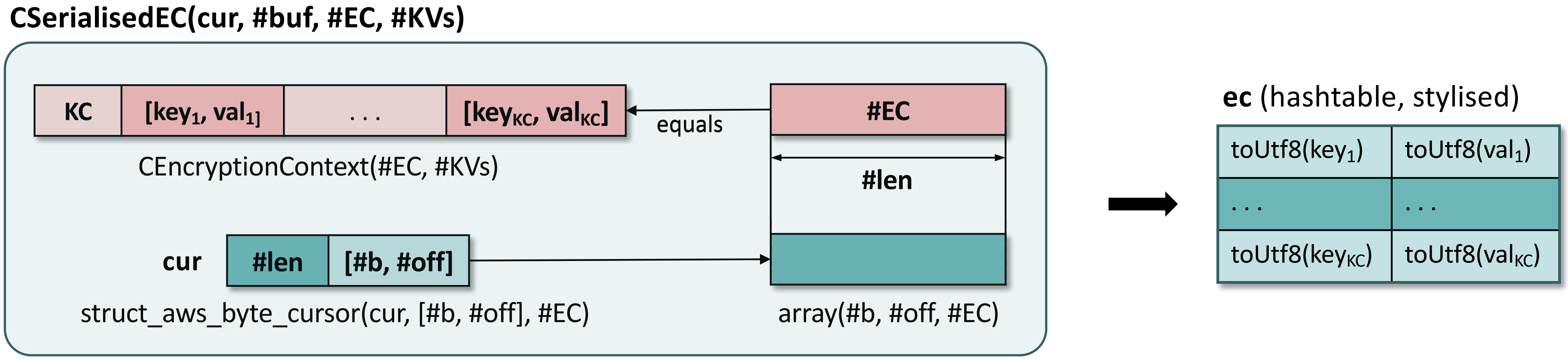
toUtf8(key$_1$)	toUtf8(val$_1$)
. . .	. . .
toUtf8(key$_{KC}$)	toUtf8(val$_{KC}$)

(frozen, with prototype null)

```
{  JSSerialisedEC(sEC, #EC, #KVs)  }
   function decodeEncryptionContext(sEC)
{  JSSerialisedEC(sEC, #EC, #KVs) ∗ JSDeserialisedEC(ret, #KVs)  }
```

C Specification: Header Deserialisation

aws_cryptosdk_enc_ctx_deserialize: deserialises the encryption context in C into a hashtable



CSerialisedEC(cur, #buf, #EC, #KVs)

KC | **[key$_1$, val$_1$]** | . . . | **[key$_{KC}$, val$_{KC}$]**

equals

#EC

#len

CEncryptionContext(#EC, #KVs)

cur | **#len** | **[#b, #off]**

struct_aws_byte_cursor(cur, [#b, #off], #EC)

array(#b, #off, #EC)

ec (hashtable, stylised)

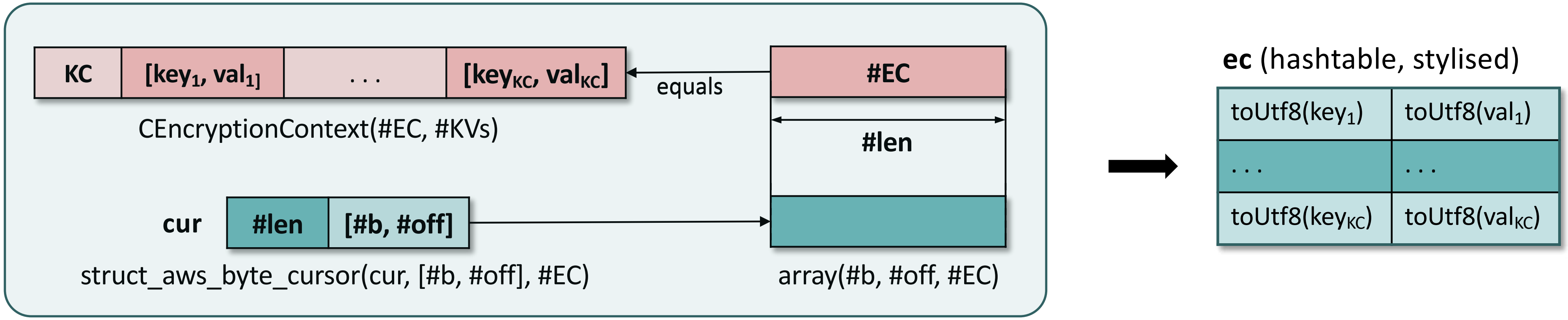
toUtf8(key$_1$)	toUtf8(val$_1$)
. . .	. . .
toUtf8(key$_{KC}$)	toUtf8(val$_{KC}$)

```
{  CSerialisedEC(cur, [#b, #off], #EC, #KVs) ∗ empty_hash_table(ec) }

   int aws_cryptosdk_enc_ctx_deserialize(
          struct aws_hash_table *ec, struct aws_byte_cursor *cur)

{  array(#b, #off, #EC) ∗ CEncryptionContext(#EC, #KVs) ∗
   aws_byte_cursor(cur, [#b, #off + l-len #EC], [ ]) ∗ CDeserialisedEC(ec, #KVs) }
```

C Specification: Header Deserialisation

aws_cryptosdk_enc_ctx_deserialize: deserialises the encryption context in C into a hashtable

CSerialisedEC(cur, #buf, #EC, #KVs)



toUtf8(key$_1$)	toUtf8(val$_1$)
. . .	. . .
toUtf8(key$_{KC}$)	toUtf8(val$_{KC}$)

```
struct aws_byte_cursor {
    size_t len;
    uint8_t *buf;
}
```

(automatically generated)

```
pred aws_byte_cursor(cur : Ptr, buf : Ptr, c : byte list) :
  struct_aws_byte_cursor(cur, #len, buf) *   // cur points to a byte cursor that views
  (buf = [#b, #off]) * array(#b, #off, c) *  // an array with contents c starting from
  (#len = l-len c)                            // offset #off in the block #b
```

Abstractions: language-dependent, resource; language-independent, pure

Caveats

- Changes to JS source code

  Original code written in TypeScript, types elided to get pure JavaScript, could be automated

  Some ES6 features rewritten to ES5 (let, const, patterns in function parameters), no expressivity loss

- Used library functions mostly axiomatised, some verified, some executed

  JS ES6 built-in libraries fully axiomatised (ArrayBuffer, DataView, etc.)

  JS ES5 built-in libraries mostly executed, a few axiomatised (Object.freeze, Array.prototype.map)

  aws-c-common library functions mostly axiomatised; a few verified with bugs discovered

- Higher-order functions either axiomatised or specialised

  The toUtf8 function is supplied as a parameter of the deserialisation module, and axiomatised as an injective function from lists of bytes to strings.

  Functions in the aws-c-common array-list library specialised for encrypted data keys

AWS: VERIFICATION

Verification Effort

Verification requires complex automatic and manual reasoning about:

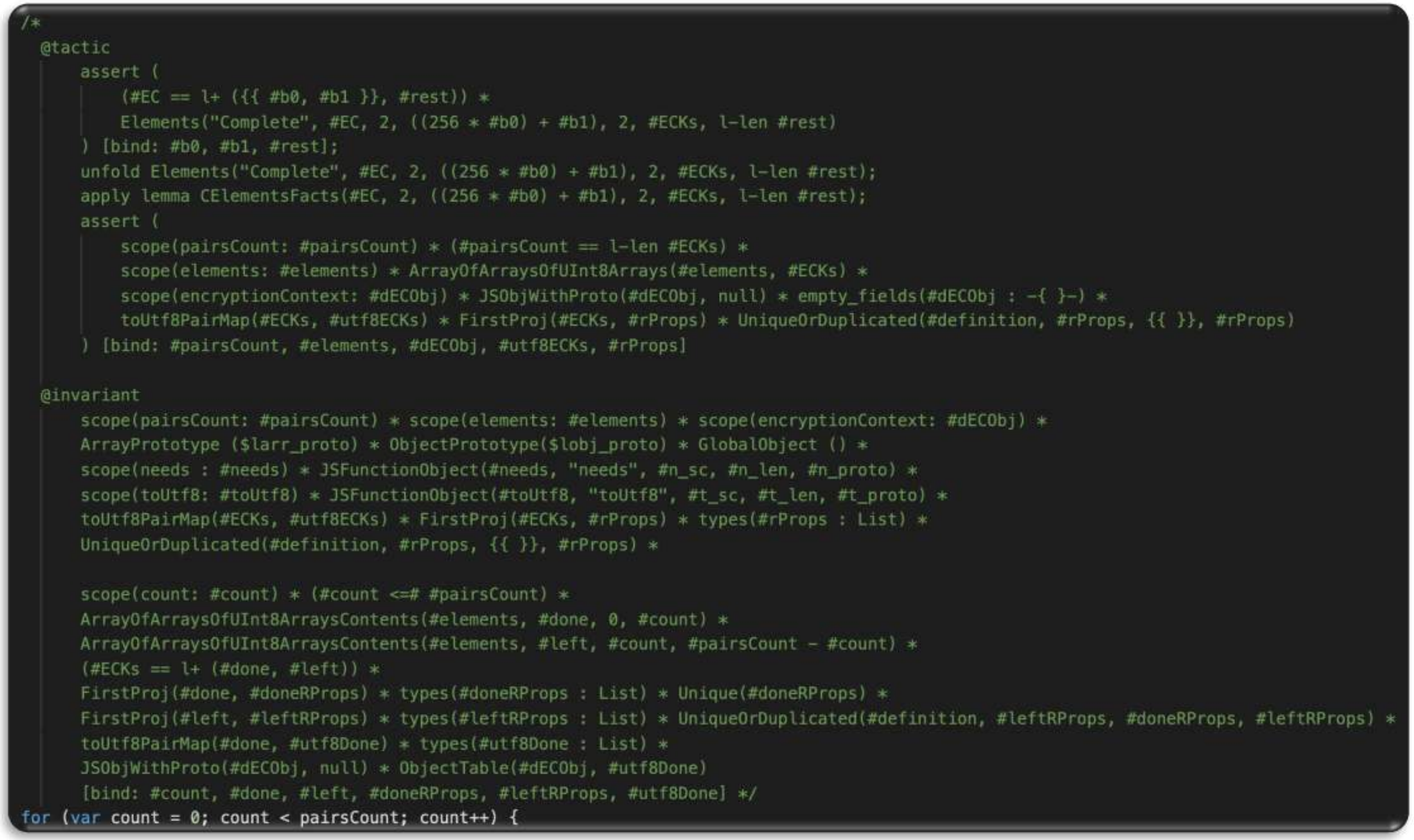
- **(A)** List concatenation and sublists with lists of symbolic size and content
- **(M)** First projection of lists of pairs
- **(M)** List element uniqueness
- **(M)** List-to-set conversion
- **(M)** Conversion to/from UTF-8
- **(M)** Manipulation of all user-defined abstractions (some unfolding, lemmas; folding is automatic)

Example of reasoning complexity

```
// Main loop of decodeEncryptionContext (JS)
// Set-up and establish loop invariant
for (var count = 0; count < pairsCount; count++) {
     var [key, value] = elements[count].map(toUtf8)
     needs(encryptionContext[key] === undefined)
     encryptionContext[key] = value
  // Re-establish invariant
}
```

Verification: decodeEncryptionContext

Set-up and establish loop invariant: 4 tactics, 27 invariant components



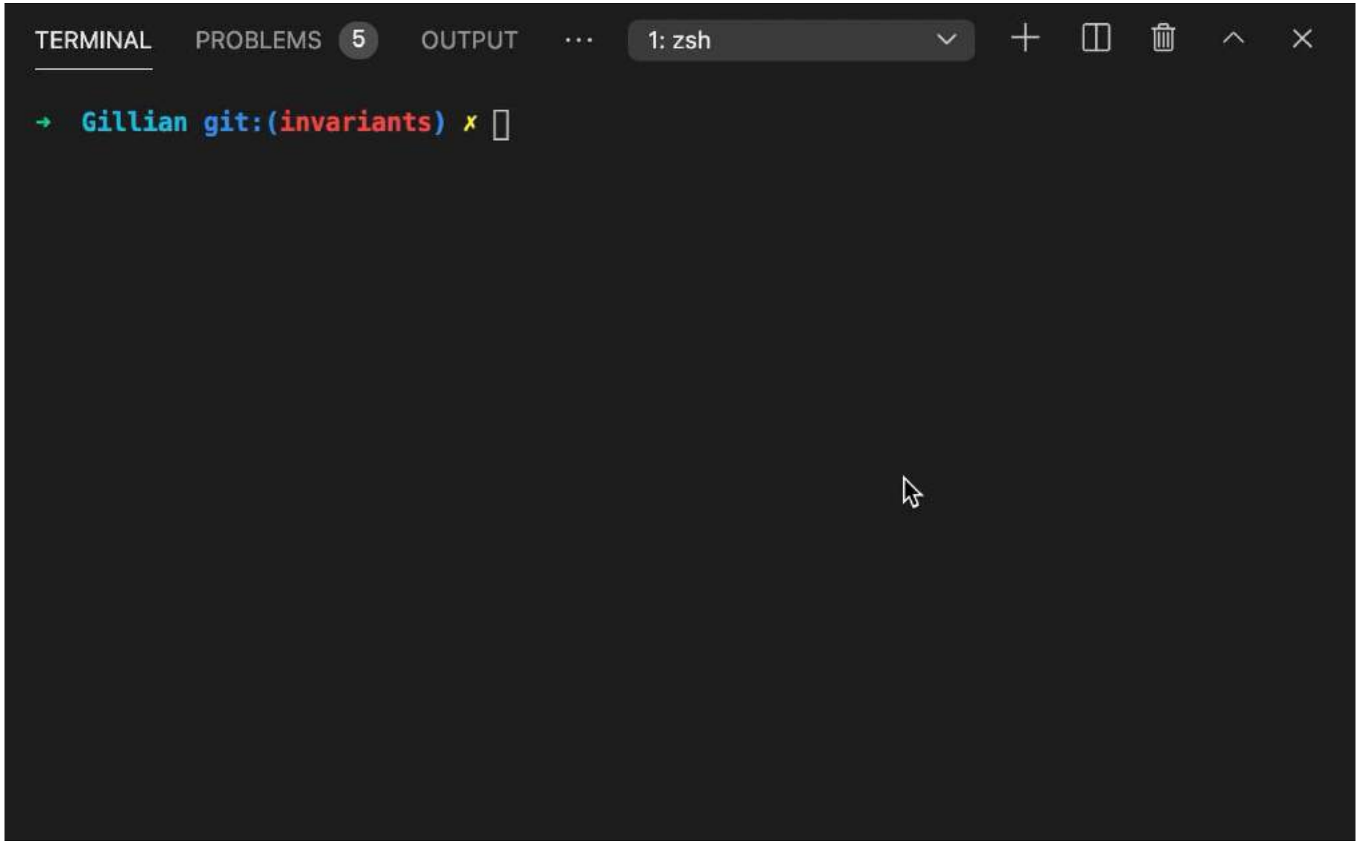
```
/*
 @tactic
     assert (
         (#EC == l+ ({{ #b0, #b1 }}, #rest)) *
         Elements("Complete", #EC, 2, ((256 * #b0) + #b1), 2, #ECKs, l-len #rest)
     ) [bind: #b0, #b1, #rest];
     unfold Elements("Complete", #EC, 2, ((256 * #b0) + #b1), 2, #ECKs, l-len #rest);
     apply lemma CElementsFacts(#EC, 2, ((256 * #b0) + #b1), 2, #ECKs, l-len #rest);
     assert (
         scope(pairsCount: #pairsCount) * (#pairsCount == l-len #ECKs) *
         scope(elements: #elements) * ArrayOfArraysOfUInt8Arrays(#elements, #ECKs) *
         scope(encryptionContext: #dECObj) * JSObjWithProto(#dECObj, null) * empty_fields(#dECObj : -{ }-) *
         toUtf8PairMap(#ECKs, #utf8ECKs) * FirstProj(#ECKs, #rProps) * UniqueOrDuplicated(#definition, #rProps, {{ }}, #rProps)
     ) [bind: #pairsCount, #elements, #dECObj, #utf8ECKs, #rProps]

 @invariant
     scope(pairsCount: #pairsCount) * scope(elements: #elements) * scope(encryptionContext: #dECObj) *
     ArrayPrototype ($larr_proto) * ObjectPrototype($lobj_proto) * GlobalObject () *
     scope(needs : #needs) * JSFunctionObject(#needs, "needs", #n_sc, #n_len, #n_proto) *
     scope(toUtf8: #toUtf8) * JSFunctionObject(#toUtf8, "toUtf8", #t_sc, #t_len, #t_proto) *
     toUtf8PairMap(#ECKs, #utf8ECKs) * FirstProj(#ECKs, #rProps) * types(#rProps : List) *
     UniqueOrDuplicated(#definition, #rProps, {{ }}, #rProps) *

     scope(count: #count) * (#count <=# #pairsCount) *
     ArrayOfArraysOfUInt8ArraysContents(#elements, #done, 0, #count) *
     ArrayOfArraysOfUInt8ArraysContents(#elements, #left, #count, #pairsCount - #count) *
     (#ECKs == l+ (#done, #left)) *
     FirstProj(#done, #doneRProps) * types(#doneRProps : List) * Unique(#doneRProps) *
     FirstProj(#left, #leftRProps) * types(#leftRProps : List) * UniqueOrDuplicated(#definition, #leftRProps, #doneRProps, #leftRProps) *
     toUtf8PairMap(#done, #utf8Done) * types(#utf8Done : List) *
     JSObjWithProto(#dECObj, null) * ObjectTable(#dECObj, #utf8Done)
     [bind: #count, #done, #left, #doneRProps, #leftRProps, #utf8Done] */
for (var count = 0; count < pairsCount; count++) {
```

Verification: decodeEncryptionContext

Re-establish loop invariant: 9 tactics

```
/*
    @tactic
        apply lemma ArrayOfArraysOfUInt8ArraysContentsAppend(#elements, #done, 0, #count);
        apply lemma IntegerLtPlusOneLe(#count, #pairsCount);
        apply lemma ObjectTableStructureAppendPVPair(#dECObj, #utf8Done, #utf8NProp, #utf8NVal);
        apply lemma toUtf8PairMapAppendPair(#done, #utf8Done, #new_prop, #new_value);
        apply lemma FirstProjAppendPair(#done, #doneRProps, #new_prop, #new_value);
        apply lemma FirstProjAppendPair(#utf8Done, #doneProps, #utf8NProp, #utf8NVal);
        apply lemma ListToSetAddElement(#doneProps, #donePropsSet, #utf8NProp);
        apply lemma UniqueAppendElement(#doneRProps, #new_prop);
        if (#definition = "Complete") then {
            unfold Unique(#leftRProps)
        }
*/
```

Actual JS Verification



```
➜  Gillian git:(invariants) ✗ ▯
```

Summary of Discovered Issues

JavaScript: Encryption Context

- if a key coincides with a property of Object.prototype, an exception is thrown erroneously*
- deserialised key-value map returned non-frozen in one scenario, allowing potential manipulation (adding/removing keys) by third parties after authentication

C: The aws-c-common Library

- over-allocation of strings: each allocated string contains eight additional, unused bytes
- undefined behaviour (adding null with 0) in the function that advances the byte cursor

* Bug predicted in the original JaVerT paper (POPL'18); found here, in cash, and in jQuery.

Gillian: Unified, Compositional Symbolic Analysis

Improving Instantiations

Gillian-While, Gillian-C, Gillian-JS

Improving Gillian

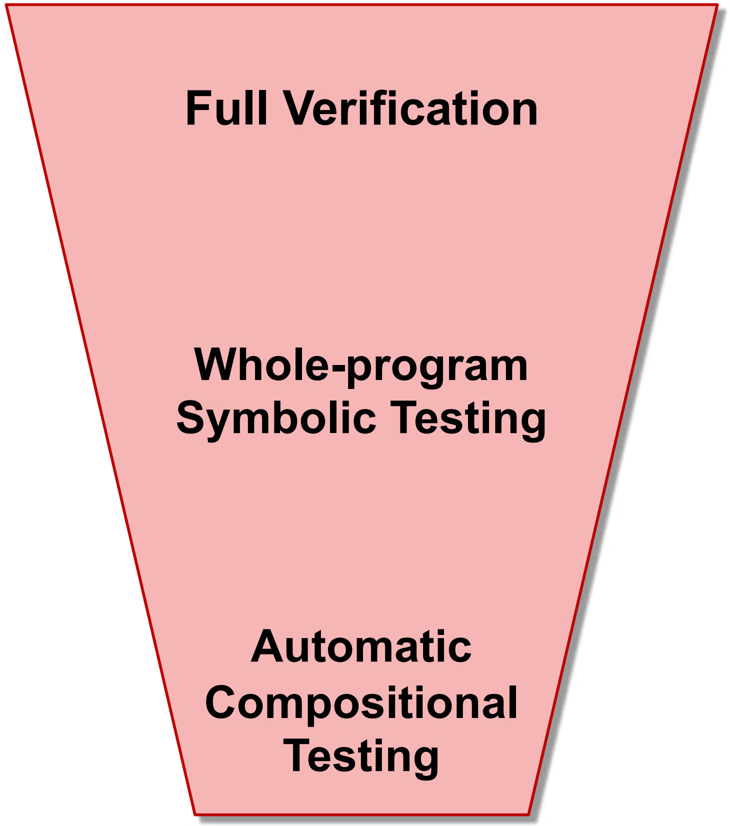
Better bi-abduction

Better first-order solver

Better error reporting

Continuous integration

Coq certification



Full Verification

Whole-program Symbolic Testing

Automatic Compositional Testing

More languages

Rust (Sacha)

WebAssembly

Various DSLs

More Analyses

Inter-operability

Concurrency

Incorrectness logic

Thank You!

Questions?