Incorrectness Logic & Under-Approximation: Foundations of Bug Detection

Azalea Raad
Imperial College London

Principles of Programming Languages
16 January 2023
State of the Art: **Correctness**

- Lots of work on *reasoning* for proving *correctness*
  - Prove the *absence of bugs*
  - *Over-approximate* reasoning
  - *Compositionality*
    - in *code* ⇒ reasoning about *incomplete components*
    - in *resources* accessed ⇒ *spatial locality*
  - *Scalability* to large teams and codebases
Hoare Logic (HL)

For all states $s$ in $p$

if running $C$ on $s$ terminates in $s'$, then $s'$ is in $q$

Hoare triples  $\{p\} \ C \ \{q\}$  iff  post($C$)$p \subseteq q$
Hoare Logic (HL)

Hoare triples \( \{p\} \ C \ \{q\} \iff \text{post}(C)p \subseteq q \)

\( q \) over-approximates \( \text{post}(C)p \)

diagram:
- true positive
- false positive
“Don’t spam the developers!”
Incorrectness Logic: A Formal Foundation for Bug Catching
Part I.
Incorrectness Logic (IL)
&
Incorrectness Separation Logic (ISL)
Incorrectness Logic (IL)

Hoare triples \( \{p\} C \{q\} \) iff \( \text{post}(C)p \subseteq q \)

For all states \( s \) in \( p \)
if running \( C \) on \( s \) terminates in \( s' \), then \( s' \) is in \( q \)

Incorrectness triples \( [p] C [q] \) iff \( \text{post}(C)p \supseteq q \)

For all states \( s \) in \( q \)
s can be reached by running \( C \) on some \( s' \) in \( p \)
Incorrectness Logic (IL)

Hoare triples \( \{p\} C \{q\} \) \iff \( \text{post}(C)p \subseteq q \)

\( q \) over-approximates \( \text{post}(C)p \)

Incorrectness triples \([p] C [q]\) \iff \( \text{post}(C)p \supseteq q \)

\( q \) under-approximates \( \text{post}(C)p \)
Incorrectness Logic (IL)

\[ [p] \ C \ [\varepsilon : q] \]

\( \varepsilon \): exit condition
- **ok**: normal execution
- **er**: erroneous execution

\[ [y=v] \ x:=y \ [\text{ok: x=y=v}] \quad [p] \ \text{error( )} \ [\text{er: p}] \]
Incorrectness Logic (IL)

\[ [p] C [\varepsilon: q] \iff \text{post}(C, \varepsilon)p \supseteq q \]

Equivalent Definition (reachability)

\[ [p] C [\varepsilon: q] \iff \forall s \in q. \exists s' \in p. (s', s) \in [C]\varepsilon \]
Short-circuiting semantics for errors
IL Proof Rules and Principles (Branches)

\[
[p] C_i [\varepsilon: q] \quad \text{some} \quad i \in \{1, 2\}
\]

\[
[p] C_1 + C_2 [\varepsilon: q]
\]

- **Drop paths/branches** (this is a **sound** under-approximation)
- **Scalable** bug detection!

\[
[p] C [\varepsilon: q] \quad \text{iff} \quad \forall s \in q. \ \exists s' \in p. \ (s',s) \in [C]\varepsilon
\]
Example

\[ y=0 \] \textbf{if} (\text{is-even}(x)) \quad \textbf{[ok: y=42 ]}

\[ y := 42 \]

\begin{align*}
[p] \ C [\varepsilon: q] \iff \forall s \in q. \ \exists s' \in p. \ (s',s) \in [C]\varepsilon
\end{align*}
Example

\[ p \]

\[
[y=0] \text{ if } (\text{is-even}(x)) \quad [\text{ok: } y=42]
\]

\[ y:= 42 \]

\[ q \]

\[ q=\{(x=0, y=42), (x=1, y=42), (x=2, y=42), \ldots\} \]

\[ [p] \ C \ [\varepsilon: q] \quad \text{iff} \quad \forall s \in q. \ \exists s' \in p. \ (s',s) \in [C]\varepsilon \]
Example

\[ p \]
\[ \begin{array}{c}
\text{if } (\text{is-even}(x)) \\
y := 42
\end{array} \]
\[ q \] \[ \begin{array}{c}
\text{[y=0]} \\
\text{[ok: y=42]}
\end{array} \]

\[ q = \{(x=0, y=42), (x=1, y=42), (x=2, y=42), \ldots\} \]

\[ [p] C [\varepsilon: q] \iff \forall s \in q. \exists s' \in p. (s',s) \in [C]\varepsilon \]
Example

\[ p \]
\[
\begin{array}{l}
\text{if } (\text{is-even}(x)) \ \\
y := 42
\end{array}
\]

\[ q \]
\[
\begin{array}{l}
\text{ok: } y = 42 \land \text{even}(x)
\end{array}
\]

\[ q = \{(x = 0, y = 42), (x = 2, y = 42), (x = 4, y = 42), \ldots\} \]

\[ [p] \ C [\varepsilon : q] \iff \forall s \in q. \exists s' \in p. (s', s) \in [C]\varepsilon \]
IL Proof Rules and Principles (Loops)

- **Bounded unrolling of loops** (this is a **sound** under-approximation)
- **Scalable** bug detection!

\[
\frac{[p] \ C^* \ \text{[ok: } p]}{[p] \ C^* \ \text{[ok: } p]} \quad \text{(Unroll-Zero)}
\]

\[
\frac{[p] \ C^*; \ C \ [\varepsilon: q]}{[p] \ C^* \ [\varepsilon: q]} \quad \text{(Unroll-Many)}
\]

\[
[p] \ C \ [\varepsilon: q] \quad \text{iff} \quad \forall s \in q. \exists s' \in p. \ (s',s) \in [C]\varepsilon
\]
IL Proof Rules and Principles (Loops continued)

∀n ∈ ℕ. [p(n)] C [ok: p(n+1)] \quad k ∈ ℕ \quad (Backwards-Variant)

\[ [p(0)] C^* [ok: p(k)] \]

- Loop **invariants** are inherently **over-approximate**
- Reason about loops **under-approximately** via **sub-variants**

\[
[p] C [\varepsilon: q] \iff \forall s \in q. \exists s' \in p. (s',s) \in [C]\varepsilon
\]
Example

\[ x=0 \quad (x++)^*; \textbf{if} \; (x==2,000,000) \text{ error; } \quad \text{[er: x=2,000,000] } \]
Example

\[ x=0 \]  \((x++)^*; \text{ if } (x==2,000,000) \text{ error}; \quad \text{[er: } x=2,000,000 \text{ ]}\]

\[ x=0 \]
\((x++)^*; \quad \text{// Backwards-Variant}\)
\[ \text{[ok: } x=2,000,000 \text{ ]}\]
\[ \text{if } (x==2,000,000) \text{ error}; \]

\[ \forall n \in \mathbb{N}. \quad [p(n)] \quad C \quad [\text{ok: } p(n+1)] \quad k \in \mathbb{N} \quad (\text{Backwards-Variant}) \]

\[ [p(0)] \quad C^* \quad [\text{ok: } p(k)] \]
Example

\[ x=0 \] \quad (x++)^*; \quad \textbf{if} \quad (x==2,000,000) \quad \textbf{error}; \quad \left[ \text{er: } x=2,000,000 \right] \checkmark

\[ x=0 \]
\quad (x++)^*; \quad // \text{ Backwards-Variant}
\quad \textbf{if} \quad (x==2,000,000) \\
\quad \text{error;} \\
\quad \left[ \text{er: } x=2,000,000 \right]

\[ \forall n \in \mathbb{N}. \quad [p(n)] \quad C \quad [\text{ok: } p(n+1)] \quad k \in \mathbb{N} \quad (\text{Backwards-Variant}) \]

\[ p(n): \quad x=n \]

\[ [\text{ok: } x=2,000,000] \]

\[ \left[ \text{er: } x=2,000,000 \right] \]
IL Proof Rules and Principles (Consequence)

\[
p' \subseteq p \quad [p'] \ C \ [\varepsilon : q'] \quad q' \supseteq q \quad (\text{Cons})
\]

- **Shrink** the post (e.g. drop disjuncts)
- **Scalable** bug detection!

\[
[p] \ C \ [\varepsilon : q] \quad \text{iff} \quad \forall s \in q. \exists s' \in p. (s',s) \in [C]_{\varepsilon}
\]
IL Proof Rules and Principles (Consequence)

\[
\frac{p' \subseteq p}{[p'] \ C \ [\varepsilon : q']} \quad q' \supseteq q \quad \text{(Cons)}
\]

\[
\frac{[p] \ C \ [\varepsilon : q_1 \lor q_2]}{[p] \ C \ [\varepsilon : q_1]}
\]

- **Shrink** the post (e.g. drop disjuncts)
- **Scalable** bug detection!

\[
[p] \ C \ [\varepsilon : q] \iff \forall s \in q. \ \exists s' \in p. \ (s', s) \in [C]_{\varepsilon}
\]
IL Proof Rules and Principles (Consequence)

\[ \frac{p' \subseteq p}{\text{[p] C [ε: q]}} \quad \frac{q' \supseteq q}{\text{[ε: q'] C [q]}} \quad \text{(Cons)} \]

\[ \frac{p' \supseteq p}{\{p'\} C \{q'\}} \quad \frac{q' \subseteq q}{\{p\} C \{q\}} \quad \text{(HL-Cons)} \]

\[ \frac{\text{[p] C [ε: q₁ ∨ q₂]}}{\text{[p] C [ε: q₁]}} \]

- **Shrink** the post (e.g. drop disjuncts)
- **Scalable** bug detection!

\[ \text{[p] C [ε: q]} \iff \forall s \in q. \exists s' \in p. (s', s) \in [C]ε \]
Incorrectness Logic: Summary

+ Under-approximate analogue of Hoare Logic

+ Formal foundation for bug catching

  – Global reasoning: non-compositional (as in original Hoare Logic)

  – Cannot target memory safety bugs (e.g. use-after-free)
Incorrectness Logic: Summary

+ Under-approximate analogue of Hoare Logic
+ Formal foundation for *bug catching*
  - Global reasoning, non-compositional (as in original Hoare Logic)
  - Cannot target memory safety bugs (e.g., use-after-free)

**Our Solution**

Incorrectness Separation Logic
What Is Separation Logic (SL)?

SL: *Local & compositional* reasoning via *ownership & separation*

❌ ideal for heap-manipulating programs with *aliasing*

```plaintext
[x] := 1;
[y] := 2;
[z] := 3;

post: \{x = 1 \land y = 2 \land z = 3\}
```
What Is Separation Logic (SL)?

SL: \textbf{Local} & \textbf{compositional} reasoning via \textit{ownership} & \textit{separation}

\begin{itemize}
\item \texttt{x} := 1;
\item \texttt{y} := 2;
\item \texttt{z} := 3;
\end{itemize}

\begin{itemize}
\item \textit{ideal for heap-manipulating programs with aliasing}
\end{itemize}

\begin{align}
\text{pre: } & \{x \neq y \land x \neq z \land y \neq z\} \\
& [x] := 1; \\
& [y] := 2; \\
& [z] := 3;
\end{align}

\text{post: } \{x = 1 \land y = 2 \land z = 3\}
What Is Separation Logic (SL)?

SL: **Local** & **compositional** reasoning via **ownership** & **separation**

→ ideal for heap-manipulating programs with **aliasing**

pre: \{ x_1 \neq x_2 \land x_1 \neq x_3 \land \ldots \}\n
\[
\begin{align*}
[x_1] &:= 1; \\
[x_2] &:= 2; \\
\ldots
\end{align*}
\]

\[ [x_n] := n; \]

post: \{ x_1 = 1 \land \ldots \land x_n = n \}
What Is Separation Logic (SL)?

SL: *Local* & *compositional* reasoning via *ownership* & *separation*

ideal for heap-manipulating programs with *aliasing*

\[
\begin{align*}
\text{pre: } & \{ x \mapsto - \ast y \mapsto - \ast z \mapsto - \} \\
& [x] := 1; \\
& [y] := 2; \\
& [z] := 3; \\
\text{post: } & \{ x \mapsto 1 \ast y \mapsto 2 \ast z \mapsto 3 \}\end{align*}
\]
What Is Separation Logic (SL)?

SL: *Local* & *compositional* reasoning via *ownership* & *separation*

*ideal for heap-manipulating programs with *aliasing***

 początku: \{ \[x\] := 1; \[y\] := 2; \[z\] := 3; \}

\[∀x,v,v'. \ x \mapsto v \ \& \ x \mapsto v' \Rightarrow false\]
The Essence of Separation Logic (SL)

**Frame Rule**

\[
\frac{\{p\} \ C \ \{q\}}{\{p \ast r\} \ C \ \{q \ast r\}}
\]

\[x \mapsto v \ast x \mapsto v' \iff \text{false} \quad \text{p \ast \text{emp} \iff p}\]
The Essence of Separation Logic (SL)

**Frame Rule**

\[
\begin{align*}
\{p\} \ C \ \{q\} \\
\{p \ast r\} \ C \ \{q \ast r\}
\end{align*}
\]

\[
\begin{align*}
x \mapsto v \ * \ x \mapsto v' \iff \text{false} \\
p \ast \ \text{emp} \iff p
\end{align*}
\]

**Local Axioms**

- **WRITE**
  \[
  \{x \mapsto -\} \ [x]:= v \ \{x \mapsto v\}
  \]

- **READ**
  \[
  \{x \mapsto v\} \ y:= [x] \ \{x \mapsto v \land y=v\}
  \]

- **ALLOC**
  \[
  \{\text{emp}\} \ x:= \text{alloc()} \ \{\exists! \ l. \ l \mapsto - \land x=l\}
  \]

- **FREE**
  \[
  \{x \mapsto -\} \ \text{free}(x) \ \{\ \text{emp} \ \}
  \]
Incorrectness Separation Logic (ISL)

IL

[p] C [ε: q]

SL

\( \{p\} C \{q\} \)

\( \{p * r\} C \{q * r\} \)

\[ x \mapsto - * x \mapsto - \equiv false \]

\[ x \mapsto v * emp \equiv x \mapsto v \]

ISL

\( [p] C [ε: q] \)

\( [p * r] C [ε: q * r] \)

\[ x \mapsto v * x \mapsto v' \equiv false \]

\[ x \mapsto v * emp \equiv x \mapsto v \]

22
ISL: Local Axioms (First Attempt)

WRITE

\[ x \mapsto v' \] \[ x := v \] \[ \text{ok: } x \mapsto v \]

\[ x = \text{null} \] \[ x := v \] \[ \text{er: } x = \text{null} \]

null-pointer dereference error
ISL: Local Axioms (First Attempt)

**WRITE**

\[ x \mapsto v' \quad [x] := v \quad \text{[ok: } x \mapsto v\text{]}

\[ x = \text{null} \quad [x] := v \quad \text{[er: } x = \text{null}\text{]}\]

*null-pointer dereference error*

**READ**

\[ x \mapsto v \quad y := [x] \quad \text{[ok: } x \mapsto v \land y = v\text{]}

\[ x = \text{null} \quad y := [x] \quad \text{[er: } x = \text{null}\text{]}\]

**ALLOC**

\[ \text{emp} \quad x := \text{alloc()} \quad \text{[ok: } \exists l. \ l \mapsto v \land x = l\text{]}\]
ISL: Local Axioms (First Attempt)

**WRITE**
\[
[x \mapsto v'] [x] := v \quad [\text{ok: } x \mapsto v]
\]
\[
[x=null] [x] := v \quad [\text{er: } x=null]
\]

*null-pointer dereference error*

**READ**
\[
[x \mapsto v] y := [x] \quad [\text{ok: } x \mapsto v \land y=v]
\]
\[
[x=null] y := [x] \quad [\text{er: } x=null]
\]

**ALLOC**
\[
[\text{emp}] x := \text{alloc()} \quad [\text{ok: } \exists l. l \mapsto v \land x=l]
\]

**FREE**
\[
[x \mapsto v] \text{free}(x) \quad [\text{ok: } \text{emp}]
\]
\[
[x=null] \text{free}(x) \quad [\text{er: } x=null]
\]
ISL: Local Axioms (First Attempt)

**WRITE**

\[
[x \mapsto v'] [x] := v \quad [\text{ok: } x \mapsto v] \\
[x=\text{null}] [x] := v \quad [\text{er: } x=\text{null}]
\]

*null-pointer dereference error*

**READ**

\[
[x \mapsto v] y := [x] \quad [\text{ok: } x \mapsto v \land y=v] \\
[x=\text{null}] y := [x] \quad [\text{er: } x=\text{null}]
\]

**ALLOC**

\[
[\text{emp}] x := \text{alloc()} \quad [\text{ok: } \exists l. l \mapsto v \land x=l]
\]

**FREE**

\[
[x \mapsto v] \text{free}(x) \quad [\text{ok: } \text{emp}] \\
[x=\text{null}] \text{free}(x) \quad [\text{er: } x=\text{null}]
\]
ISL: Local Axioms (First Attempt)

\[
\begin{align*}
\text{ISL} & \quad [p] \ C \ [\varepsilon : q] \\
& \quad \frac{}{[p \ast r] \ C \ [\varepsilon : q \ast r]} \quad x \mapsto v \ast x \mapsto v' \iff \text{false} \\
& \quad \quad \quad \quad \quad \text{emp} \ast p \iff p
\end{align*}
\]

\[\{x \mapsto v\} \text{ free}(x) \quad [\text{ok: emp}]\]

\[
\begin{align*}
[p] \ C \ [\varepsilon : q] & \quad \text{iff} \quad \forall s \in q. \ \exists s' \in p. \ (s',s) \in [C]\varepsilon
\end{align*}
\]
ISL: Local Axioms (First Attempt)

\[
\begin{align*}
\text{ISL} & \quad [p] \ C [\varepsilon : q] \\
\frac{}{[p \ast r] \ C [\varepsilon : q \ast r]} & \quad x \mapsto v \ast x \mapsto v' \Leftrightarrow \text{false} \\
& \quad \text{emp} \ast p \Leftrightarrow p
\end{align*}
\]

\[
\begin{align*}
[x \mapsto v] \text{ free}(x) [\text{ok: emp}] & \quad \text{(Frame)} \\
\frac{}{[x \mapsto v \ast x \mapsto v] \text{ free}(x) [\text{ok: emp} \ast x \mapsto v]}
\end{align*}
\]

\[
[p] \ C [\varepsilon : q] \quad \text{iff} \quad \forall s \in q. \exists s' \in p. \ (s',s) \in [C]\varepsilon
\]
ISL: Local Axioms (First Attempt)

\[
\begin{align*}
[p] C [\varepsilon : q] & \quad x \mapsto v \rightarrow x \mapsto v' \iff \text{false} \\
[p \ast r] C [\varepsilon : q \ast r] & \quad \text{emp} \ast p \iff p
\end{align*}
\]

\[
\begin{align*}
[x \mapsto v] \text{ free}(x) [\text{ok: emp}] & \\
[x \mapsto v \ast x \mapsto v] \text{ free}(x) [\text{ok: emp} \ast x \mapsto v] & \quad \text{(Frame)} \\
[\text{false}] \text{ free}(x) [\text{ok: x \mapsto v}] & \quad \text{(Cons)}
\end{align*}
\]

\[
[p] C [\varepsilon : q] \iff \forall s \in q. \exists s' \in p. (s',s) \in [C]\varepsilon
\]
ISL: Local Axioms (First Attempt)

\[
\begin{align*}
[p] C [\varepsilon: q] & \quad x \mapsto v \ast x \mapsto v' \iff \text{false} \\
[p \ast r] C [\varepsilon: q \ast r] & \quad \text{emp} \ast p \iff p
\end{align*}
\]

\[
\begin{align*}
[x \mapsto v] \text{ free}(x) [\text{ok: emp}] & \\
[x \mapsto v \ast x \mapsto v] \text{ free}(x) [\text{ok: emp} \ast x \mapsto v] & \quad \text{(Frame)}
\end{align*}
\]

\[
\begin{align*}
[x \mapsto v] \text{ free}(x) [\text{ok: emp}] & \\
[x \mapsto v \ast x \mapsto v] \text{ free}(x) [\text{ok: emp} \ast x \mapsto v] & \quad \text{(Cons)}
\end{align*}
\]

\[
\begin{align*}
[p] C [\varepsilon: q] & \quad \text{iff} \quad \forall s \in q. \exists s' \in p. (s',s) \in [C]\varepsilon
\end{align*}
\]

\[
\begin{align*}
[\text{false}] C [\varepsilon: q] & \quad \text{x} \quad \text{(unless q \Rightarrow false)}
\end{align*}
\]
ISL: Local Axioms (First Attempt)

\[
\begin{align*}
[p] & \ C \ [\epsilon : q] \\
[p \ast r] & \ C \ [\epsilon : q \ast r]
\end{align*}
\]

\[
x \mapsto v \ast x \mapsto v' \Rightarrow \text{false} \\
\text{emp} \ast p \Rightarrow p
\]

Solution:
Track Decompiled Locations!

\[
[p] \ C \ [\epsilon : q] \iff \forall s \in q. \exists s' \in p. \ (s', s) \in [C]\epsilon
\]

\[
[\text{false}] \ C \ [\epsilon : q] \times \text{(unless } q \Rightarrow \text{false)}
\]
Solution: Track Deallocated Locations!

\[ [x \mapsto v] \ free(x) \ [\text{ok: } x \not\mapsto] \]

\(x\) is \textit{deallocated}

\[
\begin{align*}
x \mapsto v & \quad \ast \quad x \mapsto v' \iff \text{false} \\
p & \quad \ast \quad \text{emp} \iff p \\
x \mapsto v & \quad \ast \quad x \not\mapsto \iff \text{false} \\
x \not\mapsto & \quad \ast \quad x \not\mapsto \iff \text{false}
\end{align*}
\]
Solution: Track Deallocated Locations!

\[ [x \mapsto v] \text{ free}(x) \ [\text{ok: } x \mapsto] \]
Solution: Track Deallocated Locations!

\[
\begin{align*}
[x \mapsto v] & \text{ free}(x) \mathrel{\iff} x \\
\hline
[x \mapsto v * x \mapsto v] & \text{ free}(x) \mathrel{\iff} x \mapsto * x \mapsto v
\end{align*}
\]
Solution: Track Deallocated Locations!

\[ [x \mapsto v] \text{ free}(x) \quad [\text{ok: } x \mapsto] \]

\[ [x \mapsto v \ast x \mapsto v] \text{ free}(x) \quad [\text{ok: } x \mapsto \ast x \mapsto v] \]

\[ \text{[false] free}(x) \quad [\text{ok: false}] \quad \checkmark \]

\[ [p] \mathcal{C} [\varepsilon: q] \quad \text{iff} \quad \forall s \in q. \exists s' \in p. (s', s) \in [\mathcal{C}]\varepsilon \]

\[ [p] \mathcal{C} [\varepsilon: \text{false}] \quad \checkmark \text{ (vacuous)} \]
ISL: Local Axioms

\[ [x \mapsto v] \text{free}(x) \quad \text{[ok: } x \mapsto ] \]

\[ [x=\text{null}] \text{free}(x) \quad \text{[er: } x=\text{null}] \]

\[ [x \mapsto ] \text{free}(x) \quad \text{[er: } x \mapsto ] \]

**double-free error**
ISL: Local Axioms

\[ [x \mapsto v] \text{ free}(x) \text{ [ok: } x \mapsto v] \]
\[ \text{FREE} \]

\[ [x \mapsto v'] [x] := v \text{ [ok: } x \mapsto v] \]
\[ \text{WRITE} \]

\[ [x \mapsto v] y := [x] \text{ [ok: } x \mapsto v \land y = v] \]
\[ \text{READ} \]

\[ [\text{emp}] x := \text{ alloc()} \text{ [ok: } \exists l. l \mapsto v \land x = l] \]
\[ \text{ALLOC} \]
ISL: Local Axioms

**FREE**

\[ x \mapsto v \] free(x) \[ \text{ok: } x \mapsto v \]  
\[ x = \text{null} \] free(x) \[ \text{er: } x = \text{null} \]

**WRITE**

\[ x \mapsto v' \] \( [x] := v \) \[ \text{ok: } x \mapsto v \]
\[ x = \text{null} \] \( [x] := v \) \[ \text{er: } x = \text{null} \]

**READ**

\[ x \mapsto v \] \( y := [x] \) \[ \text{ok: } x \mapsto v \land y = v \]
\[ x = \text{null} \] \( y := [x] \) \[ \text{er: } x = \text{null} \]

**ALLOC**

\[ \text{emp} \] \( x := \text{alloc}() \) \[ \text{ok: } \exists l. l \mapsto v \land x = l \]
\[ y \mapsto \] \( x := \text{alloc}() \) \[ \text{ok: } y \mapsto v \land x = y \]
ISL Summary

- Incorrectness \textit{Separation} Logic (ISL)
  - IL + SL for \textit{compositional bug catching}
  - \textit{Under-approximate} analogue of SL
  - Targets \textit{memory safety bugs} (e.g. use-after-free)

- Combining IL+SL: not straightforward
  - \textit{invalid frame} rule!

- Fix: a \textit{monotonic model} for frame preservation

- Recovering the \textit{footprint property} for completeness

- ISL-based \textit{analysis}
  - \textit{No-false-positives theorem:}
    All bugs found are true bugs
Part II.
Pulse-X: ISL for Scalable Bug Detection
Pulse-X at a Glance

- **Automated** program analysis for **memory safety errors** (NPEs, UAFs) and **leaks**
- Underpinned by ISL (under-approximate) — **no false positives**
- **Inter-procedural** and **bi-abductive** — under-approximate analogue of Infer
- **Compositional** (begin-anywhere analysis) — important for CI
- Deployed at Meta

**Performance**: comparable to Infer, though merely an academic tool!

**Fix rate**: comparable or better than Infer!

**Three dimensional scalability**

- code size (large codebases)
- people (large teams, CI)
- speed (high frequency of code changes)
Compositional, Begin-Anywhere Analysis

- **Analysis result** of a program = analysis results of its **parts**
  + a **method** of combining them

- **Parts**: Procedures

- **Method**: under-approximate bi-abduction

- **Analysis result**: incorrectness triples (under-approximate specs)
Pulse-X Algorithm: Proof Search in ISL

- Analyse each procedure $f$ in isolation, find its summary (collection of ISL triples)
  - A summary table $T$, initially populated only with local (pre-defined) axioms
  - Use bi-abduction and $T$ to find the summary of $f$
  - Recursion: bounded unrolling
  - Extend $T$ with the summary of $f$

- Similar bi-abductove mechanism to Infer, but:
  - Can soundly drop execution paths/branches
  - Can soundly bound loop unrolling
1. `int ssl_excert_prepend(...){
2.   SSL_EXCERT *exc= app_malloc(sizeof(*exc), "prepend cert");
3.   memset(exc, 0, sizeof(*exc));
   ...
}

Pulse-X: Null Pointer Dereference in OpenSSL

```c
1. int ssl_excert_prepend(...){
2.   SSL_EXCERT *exc= app_malloc(sizeof(*exc), "prepend cert");
3.   memset(exc, 0, sizeof(*exc));
    ...
}
```
calls CRYPTO_malloc (a malloc wrapper)
Pulse-X: Null Pointer Dereference in OpenSSL

1. `int ssl_excert_prepend(...){
2.   SSL_EXCERT *exc = app_malloc(sizeof(*exc), "prepend cert");
3.   memset(exc, 0, sizeof(*exc));
   ...
Pulse-X: Null Pointer Dereference in OpenSSL

1. \texttt{int ssl_excert_prepend(...)}{
2. \texttt{SSL_EXCERT *exc= \textcolor{blue}{app_malloc(sizeof(*exc), "prepend cert");}}
3. \texttt{memset(exc, 0, sizeof(*exc));}
   
   \textbf{null pointer dereference}

\textbf{calls CRYPTO_malloc (a malloc wrapper)}

\textbf{CRYPTO_malloc may return null!}

\begin{align*}
\text{[emp]} & \ast exc= \textcolor{blue}{app_malloc(sz, \ldots)} & \textbf{[ok: exc = null ]} \\
& + & \text{[exc = null]} \ \texttt{memset(exc,-,-)} & \textbf{[er: exc = null ]} \\
& \text{[emp]} & \texttt{ssl_excert_prepend(...)} & \textbf{[er: exc = null ]}
\end{align*}
Pulse-X: Null Pointer Dereference in OpenSSL

```c
static int ssl_x509_prepend(SSL_EXCERT **pexc)
{
    SSL_EXCERT *exc = app_malloc(sizeof(*exc), "prepend cert");
    if (!exc) {
```

@paulidale [13 days ago] Contributor

False positive, app_malloc() doesn't return if the allocation fails.

@lequangloc [13 days ago] Author

Our tool recognizes app_malloc() in test/testutil/apps_mem.c rather than the one in apps/lib/apps.c. While the former doesn't return if the allocation fails, the latter does. How do we know which one is actually called?

@paulidale [13 days ago] Contributor

It would need to look at the link lines or build dependencies to figure out which sources were used.

We should fix the one in test/testutil/apps_mem.c.
Pulse-X: Null Pointer Dereference in OpenSSL

```
956       956  @@ static int ssl_excert_prepend(SSL_EXCERT **pexc)
957       957       {
958       958          SSL_EXCERT *exc = app_malloc(sizeof(*exc), "prepend cert");
959       959          if (!exc) {
```

- **paulidale** 13 days ago  Contributor

  False positive, `app_malloc()` doesn’t return if the allocation fails.

- **lequangloc** 13 days ago  Author

  Our tool recognizes `app_malloc()` in `test/testutil/apps_mem.c` rather than the one in `apps/lib/apps.c`. While the former doesn’t return if the allocation fails, the latter does. How do we know which one is actually called?

- **paulidale** 13 days ago  Contributor

  It would need to look at the link lines or build dependencies to figure out which sources were used.

  We should fix the one in `test/testutil/apps_mem.c`.

Created pull request #15836 to commit the fix.
No False Positives: Report *All* Bugs Found?

Not quite…
1. `void foo(int *x) {
2.       *x = 42;
3. }

WRITE [x=null] *x = v [er: x=null]

[x=null] foo(x) [er: x=null]
Pulse-X: Bug Reporting

1. `void foo(int *x) {`
2. `    *x = 42;`
   `}`

```c
WRITE [x=null] *x = v [er: x=null]
```

[x=null] foo(x) [er: x=null]

```
Should we report this NPD?
```

yes

no

“But I never call foo with null!”

“Which bugs shall I report then?”

Developer

Pulse-X
Pulse-X: Bug Reporting

```c
1. void foo(int *x) {
2.   *x = 42;
}
```

**Problem**

Must consider the **whole program** to decide whether to report

**Solution**

Manifest Errors

“Developer: But I never call foo with null!”

“Pulse-X: Which bugs shall I report then?”
Pulse-X: **Manifest** Errors

- **Intuitively:** the error occurs for **all input states**

- **Formally:** $[p] C [er: q]$ is manifest iff:
  \[
  \forall s. \exists s'. (s, s') \in [C]_{er} \land s' \in (q \ast true)
  \]

- **Algorithmically:** $[p] C [er: q]$ is manifest if when: $q = \exists X. h_q \land \pi_q$:
  - $p = emp \land true$
  - $sat(q)$ and $locs(q) \subseteq X$
  - for all $\vec{v}$: $sat(\vec{v}/flv(q) \cup \vec{X})$
Pulse-X: **Manifest** Errors

- **Intuitively:** the error occurs for all input states

- **Formally:** $[p] \ C \ [er: q]$ is manifest iff:

**Theorem 3.5 (Manifest errors).** An error triple $\models [p] \ C \ [er: q]$ with $q \triangleq \exists X_q. \ \kappa_q \land \pi_q$ denotes a manifest error if:

1. $p \equiv \text{emp} \land \text{true}$;
2. $\text{sat}(q)$ holds;
3. $\text{locs}(\kappa_q) \subseteq \overrightarrow{X_q}$, where $\text{locs}(.)$ is as defined below; and
4. for all $\overrightarrow{v}$, $\text{sat}(\pi_q[\overrightarrow{v} / \overrightarrow{Y} \cup \text{locs}(\kappa_q)])$ holds, where $\overrightarrow{Y} = \text{flv}(q)$.

$\text{locs}(\text{emp}) \triangleq \emptyset \quad \text{locs}(X \mapsto X) \triangleq \{x\} \quad \text{locs}(X \mapsto V) = \text{locs}(X \not\mapsto) \triangleq \{X\} \quad \text{locs}(\kappa_1 \ast \kappa_2) \triangleq \text{locs}(\kappa_1) \cup \text{locs}(\kappa_2)$
Pulse-X: Null Pointer Dereference in OpenSSL

1. `int ssl_excert_prepend(...) {`
2. `SSL_EXCERT *exc = app_malloc(sizeof(*exc), "prepend cert");`
3. `memset(exc, 0, sizeof(*exc));`
   
   ...  
   
   null pointer dereference
   
   calls CRYPTO_malloc (a malloc wrapper)
   
   CRYPTO_malloc may return null!

   [emp] ssl_excert_prepend(...) [er: exc = null ]

Manifest Error (all calls to ssl_excert_prepend can trigger the error)!
Pulse-X: *Latent* Errors

An error triple $[p] C [er: q]$ is latent iff it is not manifest
1. int chopup_args(ARGS *args,...) {
   ...
2.   if (args->count == 0) {
3.       args->count=20;
4.       args->data= (char**)ssl_excert_prepend(...);
5.   }
6.   for (i=0; i<args->count; i++) {
7.       args->data[i]=NULL;
8.   }
9.   ...
}
int chopup_args(ARGS *args, ...){
   ...
   if (args->count == 0 ) {
      args->count = 20;
      args->data = (char**)ssl_excert_prepend(...);
   }
   for (i=0; i<args->count; i++) {
      args->data[i] = NULL;
      ...
   }
}
1. `int chopup_args(ARGS *args, ...){
    ...
2.     if (args->count == 0 ) {
3.         args->count=20;
4.         args->data= (char**)ssl_excert_prepend(...);
5.     }
6.     for (i=0; i<args->count; i++) {
    ...
7.         args->data[i]=NULL;
8.     }
9. }

Latent Error:
only calls with `args->count==0` can trigger the error
static int www_body(...) {
    ...
    io = BIO_new(BIO_f_buffer());
    ssl_bio BIO_new(BIO_f_ssl());
    ...
    BIO_push(io, ssl_bio);
    ...
    BIO_free_all(io);
    ...
    return ret;
}
static int www_body(...) {
    ...
    io = BIO_new(BIO_f_buffer());
    ssl_bio BIO_new(BIO_f_ssl());
    ...
    BIO_push(io, ssl_bio);
    ...
    BIO_free_all(io);
    ...
    return ret;
}
static int www_body(...) {
  ...
  io = BIO_new(BIO_f_buffer());
  ssl_bio BIO_new(BIO_f_ssl());
  ...
  BIO_push(io, ssl_bio);
  ...
  BIO_free_all(io);
  ...
  return ret;
}

does nothing when io is null

leaks ssl_bio
Pulse-X: Memory Leak in OpenSSL

```c
static int www_body(...) {
    ...
    io = BIO_new(BIO_f_buffer());
    ssl_bio BIO_new(BIO_f_ssl());
    ...
    BIO_push(io, ssl_bio);
    ...
    BIO_free_all(io);
    ...
    return ret;
}
```

426 lines of complex code: `io` manipulated by several procedures and multiple loops

Pulse-X performs under-approximation with bounded loop unrolling

leaks `ssl_bio`

does nothing when `io` is null
No-False Positives: Caveat

- Unknown procedures (e.g. where the code is unavailable) are treated as `skip`.
- Incomplete arithmetic solver

<table>
<thead>
<tr>
<th>Speed</th>
<th>Precision</th>
</tr>
</thead>
<tbody>
<tr>
<td>(fast but simplistic)</td>
<td>(slow but accurate)</td>
</tr>
</tbody>
</table>
No-False Positives: Caveat

- Unknown procedures (e.g. where the code is unavailable) are treated as skip
- Incomplete arithmetic solver

**Speed**

(fast but simplistic)  

**Precision**

(slow but accurate)

“Scientists seek perfection and are idealists. ... An engineer’s task is to not be idealistic. You need to be realistic as you have to compromise between conflicting interests.”
Conclusions

❖ Incorrectness Separation Logic (ISL)
  ➡ Combining IL and SL for *compositional bug catching* (in sequential programs)
  ➡ *no-false-positives* theorem

❖ Pulse-X
  ➡ Automated program analysis for detecting memory safety errors and leaks
  ➡ Manifest errors (underpinned by ISL): no false positives*
  ➡ compositional, scalable, begin-anywhere

Thank You for Listening!

azalea@imperial.ac.uk
SoundAndComplete.org
@azalearaad
Incorrectness Logic & Under-Approximation: Foundations of Bug Detection

Azalea Raad
Imperial College London

Principles of Programming Languages
16 January 2023
Part III.

ISL Extension:
Concurrent Incorrectness Separation Logic (CISL)
&
Concurrent Adversarial Separation Logic (CASL)
&
Incorrectness Non-Termination Logic (INTL)
Extension 1: Concurrent Incorrectness Separation Logic (CISL)

ISL

\[
\begin{align*}
[p] & \quad C \quad [\varepsilon : q] \\
[p \ast r] & \quad C \quad [\varepsilon : q \ast r]
\end{align*}
\]

CSL

\[
\begin{align*}
\{p_1\} & \quad C_1 \quad \{q_1\} \quad \{p_2\} \quad C_2 \quad \{q_2\} \\
\{p_1 \ast p_2\} & \quad C_1 \parallel C_2 \quad \{q_1 \ast q_2\}
\end{align*}
\]

CISL

\[
\begin{align*}
[p_1] & \quad C_1 \quad [\varepsilon : q_1] \\
[p_2] & \quad C_2 \quad [\varepsilon : q_2] \\
[p_1 \ast p_2] & \quad C_1 \parallel C_2 \quad [\varepsilon : q_1 \ast q_2]
\end{align*}
\]
Which CISL?

CSL (Correctness) Family Tree…

Graph courtesy of Ilya Sergey
Which CISL?

CSL (Correctness) Family Tree…

---

Pitfall

The Next 700 Concurrent Separation Logics

---

Graph courtesy of Ilya Sergey
Which CISL?

Pitfall

The Next 700
Concurrent *Incorrectness* Separation Logics

Graph courtesy of Ilya Sergey
Which CISL?

Solution

CISL: *general, parametric* framework that can be *instantiated* for different use cases

à la Views [Dinsdale-Young et al., 2013]
CISL Framework

- **First** unifying framework for *concurrent under-approximate* reasoning
- **General** framework for multiple bug catching analyses
  - Memory safety errors (e.g. null-pointer exception, use-after-free errors): CISL$_SV$
  - Races: CISL$_RD$
  - Deadlocks: CISL$_DD$

- **Sound**: *no false positives* (NFP) guaranteed
- Underpins **scalable** bug-catching tools (NFP for free)
  - CISL$_RD$: analogous to *RacerD* @Meta
  - CISL$_DD$: analogous to *DLTool* @Meta

- **Caveat**: cannot detect bugs where there are control flow dependencies between threads
Three Faces of Concurrency Bugs:

1. **Local** Bugs

What are they?

- They are *due to one thread*

$$\text{free}(x); \quad \text{L:}[x]:=1 \quad \| \quad C$$

local use-after-free (memory safety) bug at $L$
Three Faces of Concurrency Bugs:

1. **Local** Bugs

What are they?

- They are **due to one thread**

```
free(x);
L: [x] := 1 || C
```

Local use-after-free (memory safety) bug at L

**Thread-local** analysis tools?

- **Existing** (sequential) tools out of the box
  
  e.g. PulseX @Meta (based on ISL)

```
CISL  [p] C₁ [er: q]
ParEr  [p] C₁ || C₂ [er: q]
```

**Short-circuiting** on errors
Three Faces of Concurrency Bugs:
2, 3. **Global** Bugs

Bug is due to two or more threads, under certain interleavings

2. **data-agnostic**: threads do not affect one another’s control flow

\[ L: \text{free}(x) \parallel L': \text{free}(x) \]

(global) data-agnostic use-after-free bug at \( L \) (\( L' \))

\[ \text{free}(x); a := [z]; [z] := 1; \text{if} \ (\star) L: [x] := 1 \]

(global) data-agnostic use-after-free bug at \( L \)
Three Faces of Concurrency Bugs:

2, 3. **Global** Bugs

Bug is due to two or more threads, under certain interleavings.

2. **data-agnostic**: threads do not affect one another’s control flow

\[ L: \text{free}(x) \parallel L': \text{free}(x) \]

(global) data-agnostic use-after-free bug at \( L \) (\( L' \))

\[ \text{free}(x); \parallel a := [z]; \]
\[ [z] := 1; \parallel \text{if } (\ast) L: [x] := 1 \]

(global) data-agnostic use-after-free bug at \( L \)

3. **data-dependent** bugs: threads do affect one another’s control flow

\[ \text{free}(x); \parallel a := [z]; \]
\[ [z] := 1; \parallel \text{if } (a=1) L: [x] := 1 \]

(global) data-dependent use-after-free bug at \( L \)
Three Faces of Concurrency Bugs:
2. Global Bugs

Thread-local analysis tools?

2. data-agnostic: threads do not affect one another’s control flow
   - encode errors as ok (no short-circuiting)
   - assumed by existing tools: RacerD, DLTool @Meta

3. data-dependent bugs: threads do affect one another’s control flow
   - not handled compositionally in CISL theory

CISL

\[
\begin{align*}
\text{[p1]} & \quad \text{C1 [ok:q1]} \quad \text{[p2]} & \quad \text{C2 [ok:q2]} \\
\text{[p1 * p2]} & \quad \text{C1 || C2 [ok:q1 * q2]} & \quad \text{Par}
\end{align*}
\]
Three Faces of Concurrency Bugs:
2, 3. **Global** Bugs

**Thread-local** analysis tools?

2. **data-agnostic**: threads do not affect one another’s control flow
   - encode *errors as ok* (no short-circuiting)
   - assumed by existing tools: RacerD, DLTool @Meta

3. **data-dependent** bugs: threads do affect one another’s control flow
   - not handled compositionally in CISL theory

---

CISL

\[
[p_1] C_1 [ok:q_1] \quad [p_2] C_2 [ok:q_2] \\
[p_1 \ast p_2] C_1 \parallel C_2 [ok:q_1 \ast q_2]
\]
Two memory accesses (reads/writes), a and b, in program C race iff

1. a and b are conflicting:
   - they are by distinct threads
   - on the same location
   - at least one of them is a write

2. they appear next to each other in an interleaving (history) of C
CISL_{RD}: Data-Agnostic Races

Two memory accesses (reads/writes), a and b, in program C \textit{race} iff

1. a and b are \textit{conflicting}:
   - they are by distinct threads
   - they are on the same location
   - at least one of them is a write

2. they appear \textit{next to each other in an interleaving} (history) of C

\begin{verbatim}
1. lock l;  4. lock l;
2. unlock l; 5. [x]:= 2;
3. [x]:= 1;  6. unlock l;
\end{verbatim}

Race between lines 3, 5 witnessed by:
\[ H = [1, 2, 4, 3, 5, 6] \]
**CISL\textsubscript{RD}: Data-Agnostic Races**

Two memory accesses (reads/writes), a and b, in program C \textit{race} iff

1. a and b are \textbf{conflicting}:
   - they are by \textit{distinct} threads
   - on the \textit{same} location
   - at least \textit{one} of them is a \textit{write}

2. they appear \textbf{next to each other in an interleaving} (history) of C

\begin{align*}
1. \text{lock } l; & \quad 1. \text{lock } l; \\
2. \text{unlock } l; & \quad 4. \text{lock } l; \\
3. [x] := 1; & \quad 5. [x] := 2; \\
\end{align*}

Race between lines 3, 5 witnessed by:

\[H = [1, 2, 4, 3, 5, 6]\]

\begin{align*}
1. \text{lock } l; & \quad 4. \text{lock } l; \\
2. [x] := 1; & \quad 5. [x] := 2; \\
3. \text{unlock } l; & \quad 6. \text{unlock } l; \\
\end{align*}

No races
Methodology:
- construct sequential histories
- analyse them for races

CISL\textsubscript{RD}

\[ [ \tau_1 \leftrightarrow [] ] \quad [ \tau_2 \leftrightarrow [] ] \]

\[ [ \tau_1 \leftrightarrow [] ] \]
1. lock \( l \);
2. unlock \( l \);
3. \([x] := 1\);

\[ [ \tau_2 \leftrightarrow [] ] \]
4. lock \( l \);
5. \([x] := 2\);
6. unlock \( l \);

\[
\begin{align*}
\text{CISL} & \Rightarrow [p_1] C_1 [\text{ok:}q_1] [p_2] C_2 [\text{ok:}q_2] \\
& \Rightarrow [p_1 \ast p_2] C_1 \parallel C_2 [\text{ok:}q_1 \ast q_2]
\end{align*}
\]
Methodology:

- construct sequential histories
- analyse them for races

CISL

\[
\begin{array}{c}
\begin{align*}
[p_1] C_1 & \text{[ok:q₁]} & [p_2] C_2 & \text{[ok:q₂]} \\
[p_1 \ast p_2] C_1 & \parallel C_2 & \text{[ok:q₁ \ast q₂]} & \text{Par}
\end{align*}
\end{array}
\]

\[
\begin{array}{c}
\begin{align*}
\left[ \tau_1 \mapsto [] \ast \tau_2 \mapsto [] \right] \\
\text{1. lock } l; \\
\text{2. unlock } l; \\
\text{3. } [x] := 1;
\end{align*}
\end{array}
\]

\[
\begin{array}{c}
\begin{align*}
\left[ \tau_2 \mapsto [] \right] \\
\text{4. lock } l; \\
\text{5. } [x] := 2; \\
\text{6. unlock } l;
\end{align*}
\end{array}
\]
CISL\textsubscript{RD}

\[
[ \tau_1 \mapsto [] \ast \tau_2 \mapsto [] ]
\]

\[\begin{align*}
&1. \text{lock } l; \\
&[\text{ok: } \tau_1 \mapsto [L(\tau_1, l)]] \\
&2. \text{unlock } l; \\
&3. [x]:= 1;
\end{align*}\]

\[\begin{align*}
&4. \text{lock } l; \\
&5. [x]:= 2; \\
&6. \text{unlock } l;
\end{align*}\]

Methodology:
- construct sequential histories
- analyse them for races

CISL

\[
\frac{[p_1] C_1 [\text{ok:} q_1] \ast [p_2] C_2 [\text{ok:} q_2]}{[p_1 \ast p_2] C_1 \parallel C_2 [\text{ok:} q_1 \ast q_2]} \text{ Par}
\]
\[ \text{CISL}_{\text{RD}}: \text{Lock Axiom} \]

\[ H' = H \cdot H [\mathbb{L}(\tau, l)] \quad H' \text{ is well-formed} \]

\[ [\tau \mapsto H] \text{ lock}_\tau l \quad [\text{ok}: \tau \mapsto H'] \]

\[ \text{CISL}_{\text{RD}} \]

H is well-formed if it respects the lock semantics:
- lock \( l \) is acquired only if it is not already held
- lock \( l \) is released by \( \tau \) only if it is already held by \( \tau \)
Methodology:

- **construct sequential histories**
- **analyse them for races**

**CISL**

\[
\begin{align*}
\text{CISL}_{RD} & \quad [\tau_1 \mapsto [] * \tau_2 \mapsto []] \\
[\tau_1 \mapsto []] & \quad 1. \text{lock } l; \\
[\text{ok: } \tau_1 \mapsto [L(\tau_1, l)]] & \quad 2. \text{unlock } l; \\
[\text{ok: } \tau_1 \mapsto [L(\tau_1, l), U(\tau_1, l)]] & \quad 3. [x] := 1; \\
[\tau_2 \mapsto []] & \quad 4. \text{lock } l; \\
[\text{5. } [x] := 2;] & \quad 6. \text{unlock } l;
\end{align*}
\]
CISL<sub>RD</sub>: Unlock Axiom

A history H is well-formed iff it respects the lock semantics:

1. lock l is acquired only if it is not already held.
2. lock l is released by τ only if it is already held by τ.

$$H' = H \uplus [U(\tau, l)]$$

H' is well-formed

[τ ↦ H] unlock<sub>τ</sub> l [ok: τ ↦ H']

RD-Unlock
CISL\textsubscript{RD}

\[ [\tau_1 \mapsto [] \ast \tau_2 \mapsto []] \]

\[ [\tau_1 \mapsto []] \]
1. lock \( l \);
[ok: \( \tau_1 \mapsto [L(\tau_1, l)] \)]
2. unlock \( l \);
[ok: \( \tau_1 \mapsto [L(\tau_1, l), U(\tau_1, l)] \)]
3. \([x] := 1;\)
[ok: \( \tau_1 \mapsto [L(\tau_1, l), U(\tau_1, l), W(\tau_1, 3, x)] \)]

\[ [\tau_2 \mapsto []] \]
4. lock \( l \);
5. \([x] := 2;\)
6. unlock \( l \);

Methodology:

\(\rightarrow\) construct sequential histories
\(\rightarrow\) analyse them for races
CISL\textsubscript{RD}: Memory Access Axioms

\[
\text{RD-Read:} \quad [\tau \mapsto H] L: a :=_\tau [x] \quad [\text{ok: } \tau \mapsto H']
\]
\[
H' = H ++ [R(\tau, L, x)]
\]

\[
\text{RD-Write:} \quad [\tau \mapsto H] L: [x] :=_\tau a \quad [\text{ok: } \tau \mapsto H']
\]
\[
H' = H ++ [W(\tau, L, x)]
\]

We do not record the values read/written
Methodology:

- construct sequential histories
- analyse them for races

\[ [\tau_1 \leftrightarrow []] \times [\tau_2 \leftrightarrow []] \]

\[
\begin{align*}
\text{1.} & \quad \text{lock } l; \\
\text{[ok: } & \quad \tau_1 \leftrightarrow [L(\tau_1, l)] \] \\
\text{2.} & \quad \text{unlock } l; \\
\text{[ok: } & \quad \tau_1 \leftrightarrow [L(\tau_1, l), U(\tau_1, l)] \] \\
\text{3.} & \quad [x]:= 1; \\
\text{[ok: } & \quad \tau_1 \leftrightarrow [L(\tau_1, l), U(\tau_1, l), W(\tau_1, 3, x)] \] \\
\text{4.} & \quad \text{lock } l; \\
\text{[ok: } & \quad \tau_2 \leftrightarrow [L(\tau_2, l)] \] \\
\text{5.} & \quad [x]:= 2; \\
\text{6.} & \quad \text{unlock } l;
\end{align*}
\]
Methodology:
- **construct sequential histories**
- **analyse them for races**
CISL

\[ \tau_1 \leftrightarrow [] \ast \tau_2 \leftrightarrow [] \]

\[
\begin{align*}
[\tau_1 \leftrightarrow []] & \quad [\tau_2 \leftrightarrow []] \\
1. \ lock \ l; & \quad 4. \ lock \ l; \\
[ok: \tau_1 \leftrightarrow [L(\tau_1, l)]] & \quad [ok: \tau_2 \leftrightarrow [L(\tau_2, l)]] \\
2. \ unlock \ l; & \quad 5. [x]:= 2; \\
[ok: \tau_1 \leftrightarrow [L(\tau_1, l), U(\tau_1, l)]] & \quad [ok: \tau_2 \leftrightarrow [L(\tau_2, l), W(\tau_2, 5, x)]] \\
3. [x]:= 1; & \quad 6. \ unlock \ l; \\
[ok: \tau_1 \leftrightarrow [L(\tau_1, l), U(\tau_1, l), W(\tau_1, 3, x)]] & \quad [ok: \tau_2 \leftrightarrow [L(\tau_2, l), W(\tau_2, 5, x), U(\tau_2, l)]]
\end{align*}
\]

Methodology:
- construct sequential histories
- analyse them for races

CISL

\[
[p_1] \ C_1 \ [ok: q_1] \ [p_2] \ C_2 \ [ok: q_2] \\
[p_1 \ast p_2] \ C_1 \ || \ C_2 \ [ok: q_1 \ast q_2] \ \\ Par
\]
CISL<sub>RD</sub>

\[
[ \tau_1 \mapsto [] * \tau_2 \mapsto [] ]
\]
\[
[ \tau_1 \mapsto [] ]
1. \text{lock } l;
[\text{ok: } \tau_1 \mapsto [L(\tau_1, l)]]
2. \text{unlock } l;
[\text{ok: } \tau_1 \mapsto [L(\tau_1, l), U(\tau_1, l)]]
3. \lbrack x\rbrack := 1;
[\text{ok: } \tau_1 \mapsto [L(\tau_1, l), U(\tau_1, l), W(\tau_1, 3, x)]]
\]
\[
[ \tau_2 \mapsto [] ]
4. \text{lock } l;
[\text{ok: } \tau_2 \mapsto [L(\tau_2, l)]]
5. \lbrack x\rbrack := 2;
[\text{ok: } \tau_2 \mapsto [L(\tau_2, l), W(\tau_2, 5, x)]]
6. \text{unlock } l;
[\text{ok: } \tau_2 \mapsto [L(\tau_2, l), W(\tau_2, 5, x), U(\tau_2, l)]]
\]
\[
[\text{ok: } \tau_1 \mapsto [L(\tau_1, l), U(\tau_1, l), W(\tau_1, 3, x)] * \tau_2 \mapsto [L(\tau_2, l), W(\tau_2, 5, x), U(\tau_2, l)]]
\]

Methodology:
- construct sequential histories
- analyse them for races

\[
\begin{array}{l}
\text{CISL} \\
[p_1] C_1 [\text{ok: } q_1] [p_2] C_2 [\text{ok: } q_2] \\
[p_1 * p_2] C_1 \parallel C_2 [\text{ok: } q_1 * q_2]
\end{array}
\]
CISL$_{RD}$: race Predicate

$\tau_1 \mapsto H_1 \* \tau_2 \mapsto H_2 \Rightarrow \text{race}(L_1, L_2, H)$ iff:

there exist $H'_1, H'_2, H', a, b$ such that:

- $a$ and $b$ are conflicting accesses
- $H_1 = H'_1 ++ [a] ++ -$ and $H_2 = H'_2 ++ [b] ++ -$
- $H = H' ++ [a, b]$
- $H'$ is a permutation of $H'_1 ++ H'_2$
- $H$ is well-formed
Methodology:

- construct sequential histories
- analyse them for races

\[
\text{CISL}_{RD} \hspace{1cm} [\tau_1 \rightarrow [] \ast \tau_2 \rightarrow []] \\
[\tau_1 \rightarrow []] \\
1. \text{lock } l; \\
[\text{ok: } \tau_1 \rightarrow [L(\tau_1, l)]] \\
2. \text{unlock } l; \\
[\text{ok: } \tau_1 \rightarrow [L(\tau_1, l), U(\tau_1, l)]] \\
3. [x]:= 1; \\
[\text{ok: } \tau_1 \rightarrow [L(\tau_1, l), U(\tau_1, l), W(\tau_1, 3, x)]] \\
[\text{ok: } \tau_1 \rightarrow [L(\tau_1, l), U(\tau_1, l), W(\tau_1, 3, x)] \ast \tau_2 \rightarrow [L(\tau_2, l), W(\tau_2, 5, x), U(\tau_2, l)]] \\
[\tau_2 \rightarrow []] \\
4. \text{lock } l; \\
[\text{ok: } \tau_2 \rightarrow [L(\tau_2, l)]] \\
5. [x]:= 2; \\
[\text{ok: } \tau_2 \rightarrow [L(\tau_2, l), W(\tau_2, 5, x)]] \\
6. \text{unlock } l; \\
[\text{ok: } \tau_2 \rightarrow [L(\tau_2, l), W(\tau_2, 5, x), U(\tau_2, l)]] \\
\]
Methodology:

- construct sequential histories
- analyse them for races

\[ \text{CISL}_{RD} \]

\[
\begin{align*}
\tau_1 \mapsto [] & \times \tau_2 \mapsto [] \\
\tau_1 \mapsto [] &\ \\
1. \text{lock } l; \ \\
\text{[ok: } \tau_1 \mapsto [L(\tau_1, l)] \text{]} \\
2. \text{unlock } l; \ \\
\text{[ok: } \tau_1 \mapsto [L(\tau_1, l), U(\tau_1, l)] \text{]} \\
3. [x] := 1; \ \\
\text{[ok: } \tau_1 \mapsto [L(\tau_1, l), U(\tau_1, l), W(\tau_1, 3, x)] \text{]} \\
\end{align*}
\]

\[
\begin{align*}
\tau_2 \mapsto [] &\ \\
4. \text{lock } l; \ \\
\text{[ok: } \tau_2 \mapsto [L(\tau_2, l)] \text{]} \\
5. [x] := 2; \ \\
\text{[ok: } \tau_2 \mapsto [L(\tau_2, l), W(\tau_2, 5, x)] \text{]} \\
6. \text{unlock } l; \ \\
\text{[ok: } \tau_2 \mapsto [L(\tau_2, l), W(\tau_2, 5, x), U(\tau_2, l)] \text{]} \\
\end{align*}
\]

\[
\begin{align*}
\text{[ok: } \tau_1 \mapsto [L(\tau_1, l), U(\tau_1, l), W(\tau_1, 3, x)] & \times \tau_2 \mapsto [L(\tau_2, l), W(\tau_2, 5, x), U(\tau_2, l)] \text{]} \\
\text{[ok: } \tau_1 \mapsto [L(\tau_1, l), U(\tau_1, l), W(\tau_1, 3, x)] & \times \tau_2 \mapsto [L(\tau_2, l), W(\tau_2, 5, x), U(\tau_2, l)] \text{]} \\
\wedge \text{race}(3, 5, [L(\tau_1, l), U(\tau_1, l), L(\tau_2, l), W(\tau_1, 3, x), W(\tau_2, 5, x)]) &\ \\
\end{align*}
\]
CISL\textsubscript{RD}

Simple yet Effective in Practice à la RacerD

Methodology:
- construct sequential histories
- analyse them for races

\[
\begin{align*}
&\quad \tau_1 \rightarrow [] \quad \tau_2 \rightarrow [] \\
&\| \quad \tau_1 \rightarrow [] \\
&\quad \tau_2 \rightarrow [] \\
\end{align*}
\]
CISL Framework

- **First** unifying framework for **concurrent under-approximate** reasoning
- **General** framework for multiple bug catching analyses
  - Memory safety errors (e.g. null-pointer exception, use-after-free errors): CISL_{SV}
  - Races: CISL_{RD}
  - Deadlocks: CISL_{DD}

- Sound: **no false positives** (NFP) guaranteed
- Underpins **scalable** bug-catching tools (NFP for free)
  - CISL_{RD}: analogous to *RacerD* @Meta
  - CISL_{DD}: analogous to *DLTool* @Meta

- Caveat: cannot detect bugs where there are **control flow dependencies** between threads
Extension 2: Concurrent Adversarial Separation Logic (CASL)

- A general framework for concurrent under-approximate reasoning
- It subsumes CISL
- Can handle both data-agnostic and data-dependent bugs
- Instantiated to detect software exploits/attacks
  - Information disclosure attacks (over stacks & heaps)
  - Buffer overflow attacks (over stacks & heaps)
  - Memory safety attacks (e.g. zero allocation)
CASL for Exploit Detection: Information Disclosure Attacks

send(c, 8);
recv(c, y);

| local sec := *;
| local w[8] := {0};
| recv(c, x);
| if (x ≤ 8)
  | z := w[x];
  | send(c, z);

adversary       vulnerable program
CASL for Exploit Detection: Information Disclosure Attacks

send(c, 8);
recv(c, y);

local sec := *;
local w[8] := {0};
recv(c, x);
if (x ≤ 8)
  z := w[x];
send(c, z);

adversary
vulnerable program

stack
0 1 2 3 4 5 6 7 8

[Diagram of stack with top arrow pointing to the left]
CASL for Exploit Detection: Information Disclosure Attacks

send(c, 8);
recv(c, y);

local \textit{sec} ::= *;

local \textit{w}[8] ::= \{0\};
recv(c, x);
if (x \leq 8)
\quad z ::= w[x];
\quad send(c, z);

\begin{tabular}{c|c}
\hline
\textbf{stack} & \textbf{top} \\
\hline
0 & \textit{sec} \\
1 & \\
2 & \\
3 & \\
4 & \\
5 & \\
6 & \\
7 & \\
8 & \\
\vdots & \\
\end{tabular}

adversary  \hspace{1cm} \textit{vulnerable program}
CASL for Exploit Detection: Information Disclosure Attacks

send(c, 8);
recv(c, y);

local sec := *
local w[8] := {0};
recv(c, x);
if (x ≤ 8)
  z := w[x];
send(c, z);

adversary      vulnerable program

stack

0  sec
1  W[7]
2  W[6]
3  W[5]
4  W[4]
5  W[3]
6  W[2]
7  W[1]
8  W[0]

...
CASL for Exploit Detection: Information Disclosure Attacks

```
send(c, 8);
recv(c, y);
local sec := *
local w[8] := {0};
recv(c, x); // 8
if (x <= 8)
  z := w[x];
send(c, z);
```

<table>
<thead>
<tr>
<th>stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>7</td>
</tr>
<tr>
<td>8</td>
</tr>
</tbody>
</table>

adversary
vulnerable program
CASL for Exploit Detection: Information Disclosure Attacks

\begin{verbatim}
send(c, 8);
recv(c, y);
local sec := *;
local w[8] := {0};
recv(c, x);  // 8
if (x \leq 8)
  z := w[x];  // sec
send(c, z);
\end{verbatim}

adversary vulnerable program

\begin{center}
\begin{tabular}{|c|}
\hline
\hspace{1cm} stack \\
\hline
0 & sec \\
1 & W[7] \\
2 & W[6] \\
3 & W[5] \\
4 & W[4] \\
5 & W[3] \\
6 & W[2] \\
7 & W[1] \\
8 & W[0] \\
\hline
\end{tabular}
\end{center}
### CASL for Exploit Detection: Information Disclosure Attacks

#### Code Snippet

```plaintext
send(c, 8);
recv(c, y); // sec
recv(c, x); // 8
if (x ≤ 8)
  z := w[x]; // sec
  send(c, z);
local sec := *;
local w[8] := {0};
```

#### Stack Diagram

- **Top**: `z`
- **Adversary**: `recv(c, y)`
- **Vulnerable Program**: `local sec := *; local w[8] := {0}; recv(c, x); if (x ≤ 8) z := w[x]; send(c, z);`
- Stack:
  - `sec`
  - `W[7]`
  - `W[6]`
  - `W[5]`
  - `W[4]`
  - `W[3]`
  - `W[2]`
  - `W[1]`
  - `W[0]`
  - `...`
send(c, 8);  
recv(c, y);  // sec

local sec := *;
local w[8] := {0};
recv(c, x);  // 8
if (x ≤ 8)
  z := w[x];  // sec
  send(c, z);

---

adversary  
vulnerable program

---

information disclosure!
CASL for Exploit Detection: Memory Safety Attacks

\[
\begin{align*}
\text{send}(c, \text{maxInt}); & \quad \text{recv}(c, s); \\
\text{if } (s \leq \text{maxInt}) & \quad y := s + 1; \\
& \quad x := \text{alloc}(y); \\
& \quad \text{L: } [x+s] := 0;
\end{align*}
\]

adversary \quad \text{vulnerable program}
CASL for Exploit Detection: Memory Safety Attacks

```
recv(c, s); // maxInt
if (s ≤ maxInt)
  y := s+1;
  x := alloc(y);
L: [x+s] := 0;
```

adversary  vulnerable program
CASL for Exploit Detection: Memory Safety Attacks

\[
\begin{align*}
\text{send}(c, \text{maxInt}); \\
\text{recv}(c, s); & \quad \text{// maxInt} \\
\text{if } (s \leq \text{maxInt}) & \\
\text{\hspace{1cm}} y := s + 1; & \quad \text{// y=0} \\
\text{\hspace{1cm}} x := \text{alloc}(y); \\
\text{\hspace{1cm}} L: [x+s] := 0;
\end{align*}
\]

adversary \quad \text{vulnerable program}
CASL for Exploit Detection: Memory Safety Attacks

```
recv(c, s);  // maxInt
if (s ≤ maxInt)
  send(c, maxInt);
  y := s+1;  // y=0
  x := alloc(y);  // x=null
L: [x+s] := 0;
```

adversary  vulnerable program
CASL for Exploit Detection: Memory Safety Attacks

send(c, maxInt);
recv(c, s);  // maxInt
if (s ≤ maxInt)
  y := s+1;  // y=0
x := alloc(y);  // x=null
L: [x+s] := 0;

adversary       vulnerable program

null pointer dereference!
Termination vs Non-Termination

- **Showing** termination is compatible with correctness frameworks:
  - Every trace of a given program must terminate
  - Inherently over-approximate

  ```
  skip + x:=1
  ```
Termination vs Non-Termination

- Showing **termination** is compatible with **correctness** frameworks:
  - Every trace of a given program must terminate
  - Inherently **over-approximate**
    
    \[
    \text{skip} + x:=1
    \]

- Showing **non-termination** compatible with **incorrectness** frameworks:
  - Some trace of a given program must not-terminate
  - Inherently **under-approximate**
    
    \[
    \text{skip} + \text{while(true)skip}
    \]
Extension 3: Incorrectness Non-Termination Logic (INTL)

- A framework for detecting non-termination bugs
- Supports unstructured constructs (goto), as well exceptions and breaks
- Reasons for non-termination:
  - Infinite loops
  - Infinite recursion
  - Cyclic goto soups
INTL Proof Rules and Principles

INTL Proof Rules

= ISL Proof Rules

+ Divergence (Non-Termination) Rules
INTL Divergence Proof Rules

[p] C [∞]

Starting from some state in p, C has a divergent trace
INTL Divergence Proof Rules (Sequencing)

\[
\dfrac{[p] \ C_1 \ [\infty]}{[p] \ C_1; \ C_2 \ [\infty]} \quad \dfrac{[p] \ C_1 \ [\text{ok: q}] \quad [q] \ C_2 \ [\infty]}{[p] \ C_1; \ C_2 \ [\infty]}\]
INTL Divergence Proof Rules (Branches)

\[ [p] C_i [\infty] \quad \text{some} \ i \in \{1, 2\} \]
\[ [p] C_1 + C_2 [\varepsilon: q] \]

- **Drop paths/branches** (this is a **sound** under-approximation)
- **Scalable** bug detection!
INTL Divergence Proof Rules (Loops)

\[
\frac{[p] \ C \ \[\infty\]}{[p] \ C^* \ \[\infty\]}
\]

\[
\frac{[p] \ C \ \[\text{ok: p}\]}{[p] \ C^* \ \[\infty\]}
\]
Conclusions

❖ CISL and CASL

➡ Combining ISL and CSL for **concurrent bug catching**
➡ **no-false-positives** theorem
➡ Race detection, deadlock detection, exploit detection

❖ INTL

➡ ISL for detecting **non-termination bugs**
➡ **Unstructured** language (goto construct)
➡ **no-false-positives** theorem
➡ Infinite loop detection
➡ Infinite recursion detection — ongoing work
➡ Cyclic goto-soup — ongoing work
Announcement

I'm hiring!

Join my team (Veritas) @Imperial College

★ Internship opportunities
★ PhD studentships
★ Post-doc positions

Thank You for Listening!

azalea@imperial.ac.uk
SoundAndComplete.org
@azalearaad