

# SEC CSL

## Security Concurrent Separation Logic

CAV 2019, New York



Gidon Ernst



Toby Murray



# Motivation: Concurrent Information Flow

```
// global shared record  
struct record { bool classified; int data; };  
struct record * rec = ...;  
  
// memory-mapped IO device register  
volatile int * const OUTPUT_REG = ...;
```

# Motivation: Concurrent Information Flow

```
// global shared record
struct record { bool classified; int data; };
struct record * rec = ...;

// memory-mapped IO device register
volatile int * const OUTPUT_REG = ...;

// thread 1: output the record
while(true) {
    lock(mutex);
    if (!rec->classified)
        *OUTPUT_REG = rec->data;
    unlock(mutex);
}
```

# Motivation: Concurrent Information Flow

```
// global shared record
struct record { bool classified; int data; };
struct record * rec = ...;

// memory-mapped IO device register
volatile int * const OUTPUT_REG = ...;

// thread 1: output the record
while(true) {
    lock(mutex);
    if (!rec->classified)
        *OUTPUT_REG = rec->data;
    unlock(mutex);
}

// thread 2: edit the record
lock(mutex);
// remove classification
rec->classified = FALSE;
// erase previous data
rec->data = 0;
unlock(mutex);
```

# Motivation: Concurrent Information Flow

```
// global shared record
struct record { bool classified; int data; };
struct record * rec = ...;

// memory-mapped IO device register
volatile int * const OUTPUT_REG = ...;

// thread 1: output the record
while(true) {
    lock(mutex);
    if (!rec->classified)
        *OUTPUT_REG = rec->data;
    unlock(mutex);
}

// thread 2: edit the record
lock(mutex);
// remove classification
rec->classified = FALSE;
// erase previous data
rec->data = 0;
unlock(mutex);
```

## Correctness Proof

Memory Safety            →    Separation Logic  
+ Mutual Exclusion        →    Concurrent SL

# Motivation: Concurrent Information Flow

```
// global shared record
struct record { bool classified; int data; };
struct record * rec = ...;

// memory-mapped IO device register
volatile int * const OUTPUT_REG = ...;

// thread 1: output the record
while(true) {
    lock(mutex);
    if (!rec->classified)
        *OUTPUT_REG = rec->data;
    unlock(mutex);
}

// thread 2: edit the record
lock(mutex);
// remove classification
rec->classified = FALSE;
// erase previous data
rec->data = 0;
unlock(mutex);
```

## Correctness Proof

- Memory Safety → Separation Logic
- + Mutual Exclusion → Concurrent SL
- + No Information Leak → SECCSL

# Motivation: Concurrent Information Flow

```
// global shared record
struct record { bool classified; int data; };
struct record * rec = ...;

// memory-mapped IO device register
volatile int * const OUTPUT_REG = ...;

// thread 1: output the record
while(true) {
    lock(mutex);
    if (!rec->classified)
        *OUTPUT_REG = rec->data;
    unlock(mutex);
}

// thread 2: edit the record
lock(mutex);
// remove classification
rec->classified = FALSE;
// erase previous data
rec->data = 0;
unlock(mutex);
```

## Invariants in SECCSL

$\exists c. \text{rec->classified} \mapsto c$

$\exists d. \text{rec->data} \mapsto d$

$\exists v. \text{OUTPUT\_REG} \mapsto v$

# Motivation: Concurrent Information Flow

```
// global shared record
struct record { bool classified; int data; };
struct record * rec = ...;

// memory-mapped IO device register
volatile int * const OUTPUT_REG = ...;

// thread 1: output the record
while(true) {
    lock(mutex);
    if (!rec->classified)
        *OUTPUT_REG = rec->data;
    unlock(mutex);
}

// thread 2: edit the record
lock(mutex);
// remove classification
rec->classified = FALSE;
// erase previous data
rec->data = 0;
unlock(mutex);
```

## Invariants in SECCSL

- $\exists c. \text{rec->classified} \mapsto c \quad \wedge \quad c :: \text{low}$
- $\exists d. \text{rec->data} \mapsto d \quad \wedge \quad d :: (c ? \text{high} : \text{low})$
- $\exists v. \text{OUTPUT\_REG} \xrightarrow{\text{low}} v$



# Motivation: Concurrent Information Flow

```
// global shared record
struct record { bool classified; int data; };
struct record * rec = ...;

// memory-mapped IO device register
volatile int * const OUTPUT_REG = ...;

// thread 1: output the record
while(true) {
    lock(mutex);
    if (!rec->classified)
        *OUTPUT_REG = rec->data;
    unlock(mutex);
}

// thread 2: edit the record
lock(mutex);
// remove classification
rec->classified = FALSE;
// erase previous data
rec->data = 0;
unlock(mutex);
```

## Verification in SECC

```
./secc examples/example.c
thread1 ... success ♥ (time: 12ms)
thread2 ... success ♥ (time: 5ms)
```

# Contribution

Goal: Proof system

- ▶ for low-level concurrent systems code
- ▶ for expressive, value-dependent information flow
- ▶ with good automation

Results

- ▶ SECCSL = CSL  $\uplus$  Non-Interference
- ▶ Soundness mechanized in Isabelle/HOL
- ▶ Prototype Verifier SECC for C
- ▶ <https://covern.org/secc>



# SECCSL Assertions

- ▶ Security-Labels  $\ell \in \{\text{low}, \dots, \text{high}\}$
- ▶  $P ::= e :: e_\ell \mid e_p \xrightarrow{e_\ell} e_v \mid \varphi \Rightarrow P \mid P \star Q \mid \exists x. P$

# SECCSL Assertions

- ▶ Security-Labels  $\ell \in \{\text{low}, \dots, \text{high}\}$
- ▶  $P ::= e :: e_\ell \mid e_p \xrightarrow{e_\ell} e_v \mid \varphi \Rightarrow P \mid P \star Q \mid \exists x. P$
- ▶ Value classification  $e :: e_\ell$  (information sources)
  - ▶  $e :: \text{low}$  *data e is public*
  - ▶  $e :: (c? \text{high} : \text{low})$  *data-dependent classification*

Label  $e_\ell$  is an expression, not a type

# SECCSL Assertions

- ▶ Security-Labels  $\ell \in \{\text{low}, \dots, \text{high}\}$
- ▶  $P ::= e :: e_\ell \mid e_p \xrightarrow{e_\ell} e_v \mid \varphi \Rightarrow P \mid P \star Q \mid \exists x. P$
- ▶ Value classification  $e :: e_\ell$  (information sources)
  - ▶  $e :: \text{low}$  *data  $e$  is public*
  - ▶  $e :: (c? \text{high} : \text{low})$  *data-dependent classification*

Label  $e_\ell$  is an expression, not a type

- ▶ Location sensitivity  $e_p \xrightarrow{e_\ell} e_v$  (information sinks)
  - ▶  $e_p \xrightarrow{\text{low}} e_v$  *location  $e_p$  is attacker-observable*
  - ▶  $e_p \xrightarrow{e_\ell} e_v$  *implies  $e_p :: e_\ell$  and  $e_v :: e_\ell$*

Proof rules keep  $e_\ell$  tied to  $e_p$

# Non-Interference via Relational Semantics

- ▶ Security: Attacker can distinguish states by low data only
- ▶ Non-interference: low-equivalence is a bisimulation

$$\begin{array}{ccccccc} (s_0, h_0) & \longrightarrow & (s_1, h_1) & \longrightarrow & (s_2, h_2) & \longrightarrow & \dots \\ \left| \equiv_{\text{low}} \right. & & \left| \equiv_{\text{low}} \right. & & \left| \equiv_{\text{low}} \right. & & \\ (s'_0, h'_0) & \longrightarrow & (s'_1, h'_1) & \longrightarrow & (s'_2, h'_2) & \longrightarrow & \dots \end{array}$$

# Non-Interference via Relational Semantics

- ▶ Security: Attacker can distinguish states by low data only
- ▶ Non-interference: low-equivalence is a bisimulation

$$\begin{array}{ccccccc} (s_0, h_0) & \longrightarrow & (s_1, h_1) & \longrightarrow & (s_2, h_2) & \longrightarrow & \dots \\ \Big| \equiv_{\text{low}} & & \Big| \equiv_{\text{low}} & & \Big| \equiv_{\text{low}} & & \\ (s'_0, h'_0) & \longrightarrow & (s'_1, h'_1) & \longrightarrow & (s'_2, h'_2) & \longrightarrow & \dots \end{array}$$

- ▶ Relational Semantics, notably

$$\begin{array}{l} (s, h), (s', h') \models e :: \text{low} \iff \llbracket e \rrbracket_s = \llbracket e \rrbracket_{s'} \\ (s, h), (s', h') \models p \xrightarrow{\text{low}} a \implies \llbracket a \rrbracket_s = \llbracket a \rrbracket_{s'} \end{array}$$

# Non-Interference via Relational Semantics

- ▶ Security: Attacker can distinguish states by low data only
- ▶ Non-interference: low-equivalence is a bisimulation

$$\begin{array}{ccccccc} (s_0, h_0) & \longrightarrow & (s_1, h_1) & \longrightarrow & (s_2, h_2) & \longrightarrow & \dots \\ \Big| \equiv_{\text{low}} & & \Big| \equiv_{\text{low}} & & \Big| \equiv_{\text{low}} & & \\ (s'_0, h'_0) & \longrightarrow & (s'_1, h'_1) & \longrightarrow & (s'_2, h'_2) & \longrightarrow & \dots \end{array}$$

- ▶ Relational Semantics, notably

$$\begin{array}{l} (s, h), (s', h') \models e :: \text{low} \iff \llbracket e \rrbracket_s = \llbracket e \rrbracket_{s'} \\ (s, h), (s', h') \models p \xrightarrow{\text{low}} a \implies \llbracket a \rrbracket_s = \llbracket a \rrbracket_{s'} \end{array}$$

- ▶ **Bisimulation:** for  $\{P\} c \{Q\}$  let  $(s, h) \equiv_{\text{low}} (s', h')$   
 $\iff \exists R. \text{ lows}(P, s) \subseteq \text{ lows}(R, s)$  and  $(s, h), (s', h') \models R$



# SEC CSL = SEC $\uplus$ CSL

$$\frac{}{\{y \mapsto a\} \quad x = *y \quad \{x = a \wedge y \mapsto a\}} \text{LOAD}$$

$$\frac{}{\{ \quad \quad \quad x \mapsto a \} \quad *x = b \quad \{x \mapsto b\}} \text{STORE}$$

$$\frac{\{b \wedge P\} \quad c_1 \quad \{Q\} \quad \{\neg b \wedge P\} \quad c_2 \quad \{Q\}}{\{ \quad \quad \quad P \} \quad \text{if}(b) \quad c_1 \quad \text{else} \quad c_2 \quad \{Q\}} \text{IF}$$

$$\frac{\{ \quad \quad \quad b \wedge P \} \quad c \quad \{ \quad \quad \quad P \}}{\{ \quad \quad \quad P \} \quad \text{while}(b) \quad c \quad \{\neg b \wedge P\}} \text{WHILE}$$

# SEC CSL = SEC $\uplus$ CSL

$$\frac{}{\{y \overset{\ell}{\mapsto} a\} \quad x = *y \quad \{x = a \wedge y \overset{\ell}{\mapsto} a\}} \text{LOAD}$$

$$\frac{}{\{x :: \ell \wedge b :: \ell \wedge x \overset{\ell}{\mapsto} a\} \quad *x = b \quad \{x \overset{\ell}{\mapsto} b\}} \text{STORE (secure)}$$

$$\frac{\{b \wedge P\} \quad c_1 \quad \{Q\} \quad \{\neg b \wedge P\} \quad c_2 \quad \{Q\}}{\{b :: \text{low} \wedge P\} \quad \text{if}(b) \quad c_1 \quad \text{else} \quad c_2 \quad \{Q\}} \text{IF (timing sensitive)}$$

$$\frac{\{b :: \text{low} \wedge b \wedge P\} \quad c \quad \{b :: \text{low} \wedge P\}}{\{b :: \text{low} \wedge P\} \quad \text{while}(b) \quad c \quad \{\neg b \wedge P\}} \text{WHILE (timing sensitive)}$$

# Thanks for your attention

SECCSL: a logic for

- ▶ low-level concurrent systems code
- ▶ expressive, value-dependent information flow

SECC: a prototype automated verifier for SECCSL

Check it out: <https://covern.org/secc>

# Backup

## Related Work

- ▶ Taint-tracking, type systems: not semantic, does not accept  $\text{if}(x == x) \{ \dots \}$  when  $x :: \text{high}$
- ▶ Banerjee, Naumann (S&P 2008):  $A(e) \iff e :: \text{low}$
- ▶ Costanzo & Shao (POST 2014): Labels attached to semantic values, fails to validate  $e :: \ell \Rightarrow f(e) :: \ell$  and  $(e_1 = e_2) \Rightarrow (e_1 :: \ell \iff e_2 :: \ell)$
- ▶ Karbyshev et al (POST 2018): timing insensitive
- ▶ Eilers et al (ESOP 2018): self-composition (concurrency?)
- ▶ Vafeiadis (MFPS 2011): soundness proof has same structure
- ▶ Murray et al (EuroS&P 2018): no pointers, lack of integration between functional and security proofs

# Invariants in SecC (concrete syntax)

```
void lock(struct mutex * m);
    _(ensures exists int v.
        OUTPUT_REG |->[low] v)
    _(ensures exists int c, int d.
        &rec->is_classified |->[low] c &&
        &rec->data |->d &&
        d :: (c ? high : low))

void unlock(struct mutex * m);
    _(requires exists int v. OUTPUT_REG |->[low] v)
    _(requires exists int c, int d.
        &rec->is_classified |->[low] c &&
        &rec->data |->d &&
        d :: (c ? high : low))
```

# Logic Case Splits

$$\frac{\{\varphi \wedge P\} \text{ c } \{Q\} \quad \{\neg\varphi \wedge P\} \text{ c } \{Q\}}{\{\varphi :: \text{low} \wedge P\} \text{ c } \{Q\}} \text{ SPLIT}$$

- ▶ Why?  $(s, s') \models \varphi \iff s \models \phi \text{ and } s' \models \phi$
- ▶ (Relational semantics can represent 2 of 4 cases only)

# Secure Entailment (for CONSEQ)

$P \xRightarrow{\text{low}} Q$  holds iff

- ▶  $(s, h), (s', h') \models P$  implies  $(s, h), (s', h') \models Q$  for all  $s, h$  and  $s', h'$ , and
- ▶  $\text{lows}(P, s) \subseteq \text{lows}(Q, s)$  for all  $s$



# Observable Locations

$$\text{low}_\ell(e :: e_\ell, s) = \emptyset$$

$$\text{low}_\ell(P \star Q, s) = \text{low}_\ell(P, s) \cup \text{low}_\ell(Q, s)$$

$$\text{low}_\ell(e_p \xrightarrow{e_\ell} e_v, s) = \begin{cases} \{\llbracket e_p \rrbracket_s\}, & \llbracket e_\ell \rrbracket_s \sqsubseteq \ell \\ \emptyset, & \text{otherwise} \end{cases}$$

# Security Property

- ▶  $\text{secure}_\ell^0(P_1, c_1, Q)$  always.
- ▶  $\text{secure}_\ell^{n+1}(P_1, c_1, Q)$  iff for all pairs of states  $(s_1, h_1), (s'_1, h'_1)$ , frames  $F$ , lock sets  $L_1$  with  $(s_1, h_1), (s'_1, h'_1) \models_\ell P_1 \star F \star \text{invs}(L_1)$  where  $(\text{run } c_1, L_1, s_1, h_1) \xrightarrow{\sigma} k$  and  $(\text{run } c_1, L_1, s'_1, h'_1) \xrightarrow{\sigma} k'$  with the same (deterministic) schedule  $\sigma$  there exists  $P_2$  and a pair of successor states with either of
  - ▶  $k = (\text{stop } L_2, s_2, h_2)$  and  $k' = (\text{stop } L_2, s'_2, h'_2)$  and  $P_2 = Q$
  - ▶  $k = (\text{run } c_2, L_2, s_2, h_2)$  and  $k' = (\text{run } c_2, L_2, s'_2, h'_2)$  with  $\text{secure}_\ell^n(P_2, c_2, Q)$

such that in both cases

- ▶  $(s_2, h_2), (s'_2, h'_2) \models_\ell P_2 \star F \star \text{invs}(L_2)$  and
- ▶  $\text{low}_\ell(P_1 \star \text{invs}(L_1), s_1) \subseteq \text{low}_\ell(P_2 \star \text{invs}(L_2), s_2)$