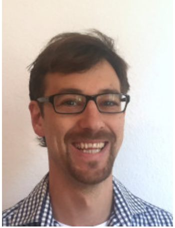
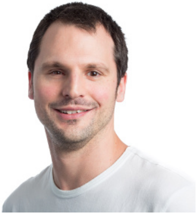
SecCSL
Security Concurrent Separation Logic

AVM 2019, Brno





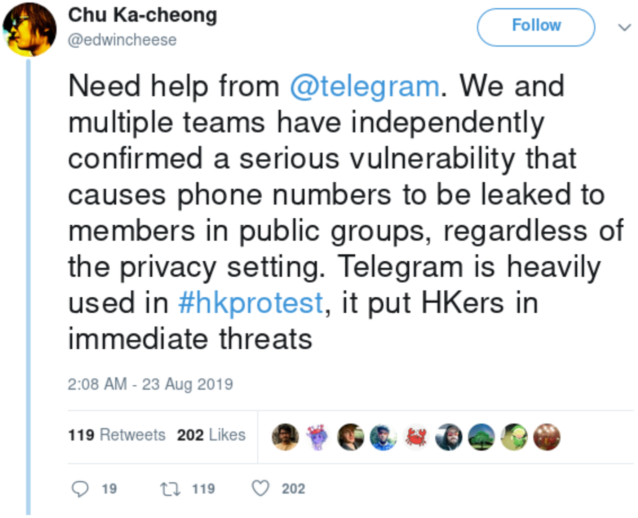
Gidon Ernst

Toby Murray

LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN

THE UNIVERSITY OF MELBOURNE

Supported by the Department of the Navy, Office of Naval Research, under award #N62909-18-1-2049.


Chu Ka-cheong
@edwincheese
Follow
Need help from @telegram. We and multiple teams have independently confirmed a serious vulnerability that causes phone numbers to be leaked to members in public groups, regardless of the privacy setting. Telegram is heavily used in #hkprotest, it put HKers in immediate threats
2:08 AM - 23 Aug 2019
119 Retweets 202 Likes
19
119
202

Noninterference [Goguen & Meseguer, S&P 1982]

Classification of data: `low` (public), `high` (secret)

Semantic criterion *Noninterference*: `high` $\not\rightsquigarrow$ `low`

Noninterference [Goguen & Meseguer, S&P 1982]

Classification of data: `low` (public), `high` (secret)

Semantic criterion *Noninterference*: `high` $\not\rightsquigarrow$ `low`

Comparison of two executions with `low`-*bisimulation*

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

Noninterference [Goguen & Meseguer, S&P 1982]

Classification of data: `low` (public), `high` (secret)

Semantic criterion *Noninterference*: `high` $\not\rightsquigarrow$ `low`

Comparison of two executions with `low`-*bisimulation*

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

$\rightarrow$ Public data does not depend on secrets

Symbolic proofs of Noninterference

$\{x :: \texttt{low} \wedge y :: \texttt{high}\}$

(1)

(2)

Symbolic proofs of Noninterference

$$\{x :: \texttt{low} \wedge y :: \texttt{high}\}$$

(1)

(2)

Relational Semantics over pairs of states

- $[\![x :: \texttt{low}]\!] \;\equiv\; (x = x')$ und $[\![x :: \texttt{high}]\!] \;\equiv\; \texttt{true}$

Symbolic proofs of Noninterference

$$\{x :: \texttt{low} \wedge y :: \texttt{high}\}$$

```
z = x + 1;
```

(1)

(2)

Relational Semantics over pairs of states

- $[\![x :: \texttt{low}]\!] \equiv (x = x')$ und $[\![x :: \texttt{high}]\!] \equiv \texttt{true}$

Symbolic proofs of Noninterference

$\{x :: \texttt{low} \wedge y :: \texttt{high}\}$

```
z = x + 1;
```
(1)

(2)

Relational Semantics over pairs of states

- $[\![x :: \texttt{low}]\!] \equiv (x = x')$ und $[\![x :: \texttt{high}]\!] \equiv \texttt{true}$
- ✓ $x = x' \implies x + 1 = x' + 1$

Symbolic proofs of Noninterference

$$\{x :: \texttt{low} \wedge y :: \texttt{high}\}$$

```
z = x + 1;
```
(1)

$$\{x :: \texttt{low} \wedge y :: \texttt{high} \wedge z :: \texttt{low}\}$$

(2)

Relational Semantics over pairs of states

- $[\![x :: \texttt{low}]\!] \equiv (x = x')$ und $[\![x :: \texttt{high}]\!] \equiv \texttt{true}$

✓ $x = x' \implies x + 1 = x' + 1$

Symbolic proofs of Noninterference

$$\{x :: \texttt{low} \wedge y :: \texttt{high}\}$$

```
z = x + 1;    (1)
```

$$\{x :: \texttt{low} \wedge y :: \texttt{high} \wedge z :: \texttt{low}\}$$

```
z = z + y;    (2)
```

Relational Semantics over pairs of states

- $[\![x :: \texttt{low}]\!] \equiv (x = x')$ und $[\![x :: \texttt{high}]\!] \equiv \texttt{true}$

✓ $x = x' \implies x + 1 = x' + 1$

Symbolic proofs of Noninterference

$\{x :: \texttt{low} \wedge y :: \texttt{high}\}$

```
z = x + 1;
```
(1)

$\{x :: \texttt{low} \wedge y :: \texttt{high} \wedge z :: \texttt{low}\}$

```
z = z + y;
```
(2)

Relational Semantics over pairs of states

- $[\![x :: \texttt{low}]\!] \equiv (x = x')$ und $[\![x :: \texttt{high}]\!] \equiv \texttt{true}$

✓ $x = x' \implies x + 1 = x' + 1$

✗ $z = z' \not\Longrightarrow z + y = z' + y'$

Symbolic proofs of Noninterference

$$\{x :: \texttt{low} \wedge y :: \texttt{high}\}$$

$$\texttt{z = x + 1;} \tag{1}$$

$$\{x :: \texttt{low} \wedge y :: \texttt{high} \wedge z :: \texttt{low}\}$$

$$\texttt{z = z + y;} \tag{2}$$

$$\{x :: \texttt{low} \wedge y :: \texttt{high} \wedge z :: \texttt{high}\}$$

Relational Semantics over pairs of states

- $[\![x :: \texttt{low}]\!] \equiv (x = x')$ und $[\![x :: \texttt{high}]\!] \equiv \texttt{true}$
- ✓ $x = x' \implies x + 1 = x' + 1$
- ✗ $z = z' \not\Longrightarrow z + y = z' + y'$

This Work

Goal: Proof system

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

Results

- SecCSL = CSL $\uplus$ Noninterference
- Soundness mechanized in Isabelle/HOL
- Prototype Verifier SecC for C
- https://covern.org/secc

Example: 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 = ...;
```

Example: 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);
}
```

Example: 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);
```

Example: 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	$\rightarrow$	Separation Logic
+	Mutual Exclusion	$\rightarrow$	Concurrent SL

Example: 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	$\rightarrow$	Separation Logic
+	Mutual Exclusion	$\rightarrow$	Concurrent SL
+	No Information Leak	$\rightarrow$	SecCSL

Example: 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

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

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

$$\exists\, v.\ \texttt{OUTPUT\_REG} \longmapsto v$$

Example: 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

$$\exists\, c.\ \texttt{rec->classified} \mapsto c \quad \land \quad c :: \texttt{low}$$

$$\exists\, d.\ \texttt{rec->data} \mapsto d \quad \land \quad d :: (c \,?\, \texttt{high} : \texttt{low})$$

$$\exists\, v.\ \texttt{OUTPUT\_REG} \overset{\texttt{low}}{\longmapsto} v$$

Example: 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)
```

SecCSL Assertions

- Security-Labels $\ell \in \{\texttt{low}, \ldots, \texttt{high}\}$
- $P ::= e :: e_\ell \mid e_p \overset{e_\ell}{\longmapsto} e_v \mid \varphi \Rightarrow P \mid P \star Q \mid \exists\, x.\ P$

SecCSL Assertions

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

- *Value* classification $e :: e_\ell$ (information sources)
  - $e :: \texttt{low}$ — *data e is public*
  - $e :: (c \mathbin{?} \texttt{high} : \texttt{low})$ — *data-dependent classification*

  Label e_ℓ is an expression, not a type

SecCSL Assertions

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

  Label e_ℓ is an expression, not a type
- *Location* sensitivity $e_p \overset{e_\ell}{\mapsto} e_v$ (information sinks)
  - $e_p \overset{\texttt{low}}{\longmapsto} e_v$ — *location e_p is attacker-observable*
  - $e_p \overset{e_\ell}{\mapsto} e_v$ — *implies $e_p :: e_\ell$ and $e_v :: e_\ell$*

  Proof rules keep e_ℓ tied to e_p

SecCSL = Sec ⊎ CSL

$$\frac{}{\{\mathsf{y} \mapsto a\}\ \ \mathtt{x = *y}\ \ \{\mathsf{x} = a \wedge \mathsf{y} \mapsto a\}}\ \textsc{Load}$$

$$\frac{}{\{\mathsf{x} \mapsto a\}\ \ \mathtt{*x = }\ b\ \ \{\mathsf{x} \mapsto b\}}\ \textsc{Store}$$

$$\frac{\{b \wedge P\}\ \ c_1\ \ \{Q\} \qquad \{\neg b \wedge P\}\ \ c_2\ \ \{Q\}}{\{P\}\ \ \mathtt{if}(b)\ c_1\ \mathtt{else}\ c_2\ \ \{Q\}}\ \textsc{If}$$

$$\frac{\{b \wedge P\}\ \ c\ \ \{P\}}{\{P\}\ \ \mathtt{while}(b)\ c\ \ \{\neg b \wedge P\}}\ \textsc{While}$$

SecCSL = Sec ⊎ CSL

$$\frac{}{\{\mathsf{y} \overset{\ell}{\mapsto} a\}\ \ \texttt{x = *y}\ \ \{\mathsf{x} = a \land \mathsf{y} \overset{\ell}{\mapsto} a\}}\ \textsc{Load}$$

$$\frac{}{\{\mathsf{x} :: \ell \land b :: \ell \land \mathsf{x} \overset{\ell}{\mapsto} a\}\ \ \texttt{*x = } b\ \ \{\mathsf{x} \overset{\ell}{\mapsto} b\}}\ \textsc{Store}\ \text{(secure)}$$

$$\frac{\{b \land P\}\ \ c_1\ \ \{Q\} \qquad \{\neg b \land P\}\ \ c_2\ \ \{Q\}}{\{b :: \texttt{low} \land P\}\ \ \texttt{if(}b\texttt{)}\ c_1\ \texttt{else}\ c_2\ \ \{Q\}}\ \textsc{If}\ \text{(timing sensitive)}$$

$$\frac{\{b :: \texttt{low} \land b \land P\}\ \ c\ \ \{b :: \texttt{low} \land P\}}{\{b :: \texttt{low} \land P\}\ \ \texttt{while(}b\texttt{)}\ c\ \ \{\neg b \land P\}}\ \textsc{While}\ \text{(timing sensitive)}$$

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
  https://covern.org/secc

Ongoing

- ring-buffer connecting multiple security domains
- VerifyThis 2020 long-term challenge: PGP keyserver
  https://verifythis.github.io
  https://github.com/gernst/verifythis2020

Backup

Related Work

- Taint-tracking, type systems: not semantic, does not accept `if(`x` == `x`) { ... }` when $x :: \texttt{high}$
- Banerjee, Naumann (S&P 2008): $\mathsf{A}(e) \iff e :: \texttt{low}$
- 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 <u>in</u>sensitive
- 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

$$\frac{\{\varphi \wedge P\} \;\; c \;\; \{Q\} \quad \{\neg\varphi \wedge P\} \;\; c \;\; \{Q\}}{\{\varphi :: \mathtt{low} \wedge P\} \;\; c \;\; \{Q\}} \textsc{Split}$$

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

Secure Entailment (for CONSEQ)

$P \overset{\texttt{low}}{\Longrightarrow} Q$ holds iff

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

Observable Locations

$$\mathrm{lows}_\ell(e :: e_\ell, s) = \emptyset$$

$$\mathrm{lows}_\ell(P \star Q, s) = \mathrm{lows}_\ell(P, s) \cup \mathrm{lows}_\ell(Q, s)$$

$$\mathrm{lows}_\ell(e_p \overset{e_\ell}{\longmapsto} e_v, s) = \begin{cases} \{[\![e_p]\!]_s\}, & [\![e_\ell]\!]_s \sqsubseteq \ell \\ \emptyset, & \text{otherwise} \end{cases}$$

Security Property

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

  such that in both cases
  - $(s_2, h_2), (s'_2, h'_2) \models_\ell P_2 \star F \star \mathrm{invs}(L_2)$ and
  - $\mathrm{lows}_\ell(P_1 \star \mathrm{invs}(L_1), s_1) \subseteq \mathrm{lows}_\ell(P_2 \star \mathrm{invs}(L_2), s_2)$