Deductive Verification of Information Flow in Security Concurrent Separation Logic

Gidon Ernst

LMU Munich, Germany gidon.ernst@lmu.de
https://www.sosy-lab.org/people/ernst/

CS Seminar, University of Konstanz, November 27, 2024
Joint work with: Toby Murray, Mukesh Tiwari, David Naumann

“40M credit cards hacked” [CNNMoney 2005]

Credit card information publicly accessible

“2 Billion Records Exposed In Massive Smart Home Device Breach” [Forbes, 2. Juli 2019]

Customer data not sufficiently protected (no password)

Facebook: Shadow Profile Leak [2015]

User profiles accessible: wrong association between users and private data

Remote timing attacks are practical [Brumley & Boneh, Usenix Security 2003]

OpenSSL
Cryptography and SSL/TLS Toolkit

Reconstruction of private keys from timing attacks

✗ Information Leak: Sensitive Source ⇝ Public Sink

```
bool check(char *input, char *password) {
    ...
    log("incorrect password: %s\n", input);
    ...
}
```

✗ Timing Leak: Runtime Depends on Sensitive Data

```
bool check(char *input, char *password) {
    return strcmp(input, password) == 0;
}
```

✗ Timing Leak: Runtime Depends on Sensitive Data

```
bool check(char *input, char *password) {
    for(int i=0; i<strlen(password); i++) {
        if(input[i] != password[i])
            return false;
    }
    return true;
}
```

✗ Timing Leak: Runtime Depends on Sensitive Data

```
bool check(char *input, char *password) {
    for(int i=0; i<strlen(password); i++) {
        if(input[i] != password[i])
            return false;
    }
    return true;
}
```

- n iterations of the loop → first n characters are correct
- Attack is linear in $O($`strlen(password)`$)$

✗ Timing Leak: Runtime Depends on Sensitive Data

```
bool check(char ∗input, char ∗password) {
    for(int i=0; i<strlen(password); i++) {
        if(input[i] != password[i])
            return false;
    }
    return true;
}
```

- n iterations of the loop → first n characters are correct
- Attack is linear in $O($`strlen(password)`$)$
- Buggy if `strlen(input) != strlen(password)`

seL4: Information Flow Enforcement [Murray+ S&P 2013]

Security. Performance. Proof.

Verification via proofs

- ✓ Full isolation between running processess
- ✓ Relies heavily on functional correctness [Klein+ SOSP 2009]

seL4: Information Flow Enforcement [Murray+ S&P 2013]

Verification via proofs

- ✓ Full isolation between running processess
- ✓ Relies heavily on functional correctness [Klein+ SOSP 2009]
- ✗ Not much proof automation (pointers, concurrency)
- ✗ No specialized tool (Isabelle/HOL)

Continuous formal verification of s2n [Chudnov+ CAV 2018]

TLS Bibliothek

- ✓ Formal specification with Cryptol [Lewis, FMSE 2007]
- ✓ Formal proof with SAW [Galois, SIG Ada 2013]
- ✓ Continuous re-verification upon changes

Continuous formal verification of s2n [Chudnov+ CAV 2018]

s2n

TLS Bibliothek

- ✓ Formal specification with Cryptol [Lewis, FMSE 2007]
- ✓ Formal proof with SAW [Galois, SIG Ada 2013]
- ✓ Continuous re-verification upon changes (currently: AWS CodeBuild unknown)

Continuous formal verification of s2n [Chudnov+ CAV 2018]

s2n

TLS Bibliothek

- ✓ Formal specification with Cryptol [Lewis, FMSE 2007]
- ✓ Formal proof with SAW [Galois, SIG Ada 2013]
- ✓ Continuous re-verification upon changes (currently: AWS CodeBuild unknown)
- ? Timing attacks? [Albrecht & Paterson, Eurocrypt 2016]
- ✗ Bounded non-modular Verification

Goal: Correctness + Security Verification for C Code

- SecCSL: Security Concurrent Separation Logic [Ernst & Murray CAV 2019]
    - Integrates security reasoning into deductive verification
    - Allows semantic reasoning about information flow

- Assume but Verify [Murray+ CCS 2023]
    - Bridges between high-level policies and code-level proofs
    - Knowledge-based characterization of declassifications

- Project Overview: `https://covern.org/`
- Tool support & case studies: `https://bitbucket.org/covern/secc`
- Formalization and proofs are mechanized in Isabelle/HOL ✓

Coin Guessing (1)

Coin Guessing (2)

```
void reveal() {
    int left  = coin();
    int right = coin();

    _(assume left :: low);
    print(left);

    print(right);
}
```

Notation x :: `low` when x has “low” sensitivity and thus is *public*

Coin Guessing (2)

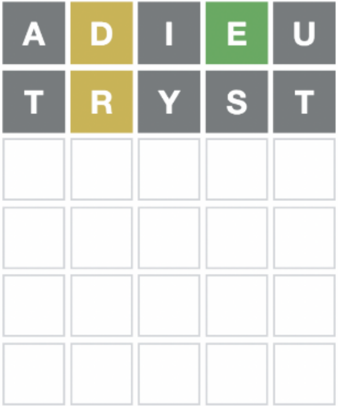
```
void reveal() {
    int left  = coin();
    int right = coin();

    _(assume left :: low);
    print(left);

    print(right);
}
```

Notation x :: `low` when x has
"low" sensitivity and thus is *public*

Some insights on "noninterference" [Goguen & Meseguer, S&P 1982]

- Information flow compares (pairs of) possible worlds
- Even simple examples require logical (semantic) reasoning

Information Security in Wordle[1]



The game is supposed to reveal only

- green: letters at correct position
- yellow: letters somewhere else in word
- (letters typed in: player knows already)

---

[1] From Toby Murray's fantastic blog posts: https://verse.systems/blog/post/2022-03-01-on-software-perfection/ https://verse.systems/blog/post/2022-03-02-specifying-wordle-secure/

Example: Deductive Proofs about Noninterference

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

Example: Deductive Proofs about Noninterference

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

Relational semantics over pairs of states

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

Example: Deductive Proofs about Noninterference

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

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

Relational semantics over pairs of states

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

Example: Deductive Proofs about Noninterference

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

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

Relational semantics over pairs of states

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

✓ $x = x' \implies x + 1 = x' + 1$

Example: Deductive Proofs about Noninterference

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

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

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

Relational semantics over pairs of states

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

✓ $x = x' \implies x + 1 = x' + 1$

Example: Deductive Proofs about Noninterference

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

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

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

```
z = z + y;
```

Relational semantics over pairs of states

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

✓ $x = x' \implies x + 1 = x' + 1$

Example: Deductive Proofs about Noninterference

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

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

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

```
z = z + y;
```

Relational semantics over pairs of states

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

✓ $x = x' \implies x + 1 = x' + 1$

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

Example: Deductive Proofs about Noninterference

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

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

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

```
z = z + y;
```

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

Relational semantics over pairs of states

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

Goal: Correctness + Security Verification for C Code

- SecCSL: Security Concurrent Separation Logic [Ernst & Murray CAV 2019]
  - Integrates security reasoning into deductive verification
  - Allows semantic reasoning about information flow

- Assume but Verify [Murray+ CCS 2023]
  - Bridges between high-level policies and code-level proofs
  - Knowledge-based characterization of declassifications

- Project Overview: `https://covern.org/`
- Tool support & case studies: `https://bitbucket.org/covern/secc`
- Formalization and proofs are mechanized in Isabelle/HOL ✓

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                 || // thread 2: edit the record
while(true) {                                  || lock(mutex);
    lock(mutex);                               || // remove classification
    if (!rec->classified)                      || rec->classified = FALSE;
        *OUTPUT_REG = rec->data;               || // erase previous data
    unlock(mutex);                             || rec->data = 0;
}                                              || unlock(mutex);
```

Correctness Proof

Memory Safety → Separation Logic [Reynolds, LICS 2002]

+ Mutual Exclusion → Concurrent SL [O'Hearn, CONCUR 2004]

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 → Separation Logic [Reynolds, LICS 2002]

\+ Mutual Exclusion → Concurrent SL [O'Hearn, CONCUR 2004]

\+ No Information Leak → 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                 || // thread 2: edit the record
while(true) {                                  || lock(mutex);
    lock(mutex);                               || // remove classification
    if (!rec->classified)                      || rec->classified = FALSE;
        *OUTPUT_REG = rec->data;               || // erase previous data
    unlock(mutex);                             || 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                  || // thread 2: edit the record
while(true) {                                   || lock(mutex);
    lock(mutex);                                || // remove classification
    if (!rec->classified)                       || rec->classified = FALSE;
        *OUTPUT_REG = rec->data;                || // erase previous data
    unlock(mutex);                              || rec->data = 0;
}                                               || unlock(mutex);
```

Invariants in SecCSL

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

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

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

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                ||  // thread 2: edit the record
while(true) {                                 ||  lock(mutex);
    lock(mutex);                              ||  // remove classification
    if (!rec->classified)                     ||  rec->classified = FALSE;
        *OUTPUT_REG = rec->data;              ||  // erase previous data
    unlock(mutex);                            ||  rec->data = 0;
}                                             ||  unlock(mutex);
```

Verification in SecC

```
./SecC.sh examples/case-studies/cav2019.c
thread1 ...  success ♥  (time:  221ms)
thread2 ...  success ♥  (time:   54ms)
```

SecCSL Assertions

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

SecCSL Assertions

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

SecCSL Assertions

- Security-Labels $\ell \in \{\texttt{low}, \dots, \texttt{high}\}$
- $P ::= e :: e_\ell \mid e_p \xmapsto{e_\ell} e_v \mid \varphi \Rightarrow P \mid P \star Q \mid \exists\, x.\, P$
- *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 $e_p \xmapsto{e_\ell} e_v$ (information sources & sinks)
  - $e_p \xmapsto{\texttt{low}} e_v$ *location e_p is attacker-observable*
  - $e_p \xmapsto{e_\ell} e_v$ *implies $e_p :: e_\ell$ and $e_v :: e_\ell$*

  Proof rules keep e_ℓ tied to e_p

SecCSL = Sec ⊎ CSL

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

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

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

$$\frac{\{\qquad\qquad b \land P\}\ c\ \{\qquad\qquad P\}}{\{\qquad\qquad P\}\ \texttt{while(}b\texttt{)}\ c\ \{\neg b \land P\}}\ \textsc{While}$$

SecCSL = Sec ⊎ CSL

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

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

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

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

Example Proof: Thread 2

```
lock(mutex);



rec->is_classified = false;
rec->data = 0;



unlock(mutex);
```

Example Proof: Thread 2

```
lock(mutex);
```

$$\{\ \texttt{rec} \mapsto (c, d) \land c :: \texttt{low} \land (\neg\, c \Rightarrow d :: \texttt{low})$$

$$\star\ \texttt{OUTPUT\_REG} \overset{\texttt{low}}{\longmapsto} v \land v :: \texttt{low}\ \}$$

```
rec->is_classified = false;
rec->data = 0;



unlock(mutex);
```

Example Proof: Thread 2

```
lock(mutex);
```

$$\{\ \mathtt{rec} \mapsto (c, d) \land c :: \mathtt{low} \land (\lnot\, c \Rightarrow d :: \mathtt{low}) \\ \star\ \mathtt{OUTPUT\_REG} \xmapsto{\mathtt{low}} v \land v :: \mathtt{low}\ \}$$

```
rec->is_classified = false;
rec->data = 0;
```

$$\{\ \mathtt{rec} \mapsto (\mathtt{false}, 0) \land \mathtt{false} :: \mathtt{low} \land (\mathtt{true} \Rightarrow 0 :: \mathtt{low}) \\ \star\ \mathtt{OUTPUT\_REG} \xmapsto{\mathtt{low}} v \land v :: \mathtt{low}\ \}$$

```
unlock(mutex);
```

Summary: Security Concurrent Separation Logic

Goals:

- Integrates security reasoning into deductive verification
- Allows semantic reasoning about information flow

Attacker model:

- has access to code & can observe timing of execution
  → trade-off taken to not allow branching on secrets
- can observe public memory locations and public outputs

Guarantee:

- Verified programs are secure wrt. these attackers . . .
- . . . in the absence of declassification

Goal: Correctness + Security Verification for C Code

- SecCSL: Security Concurrent Separation Logic [Ernst & Murray CAV 2019]
  - Integrates security reasoning into deductive verification
  - Allows semantic reasoning about information flow

- Assume but Verify [Murray+ CCS 2023]
  - Bridges between high-level policies and code-level proofs
  - Knowledge-based characterization of declassifications

- Project Overview: `https://covern.org/`
- Tool support & case studies: `https://bitbucket.org/covern/secc`
- Formalization and proofs are mechanized in Isabelle/HOL ✓

Motivation: High-level Declassification Policies

Contribution (so far) ✓ Logic + soundness ✓ Tool + small examples

Motivation for follow-up work:

- Semantics of modular verification unclear [Chudnov & Naumann 2018]
  `assume` $x :: \ell$ and `assert` $x :: \ell$ cannot be explained with a single execution
- “when” to release “what” information [Sabelfeld & Sands 2009]
  Want high-level declassification policies
- Opportunity to get more experience on larger case-studies

Declassification via `assume`

```
int get_balance(int pin, int guess)
    _(requires (pin == guess) :: low)
    _(requires guess :: low)
    _(ensures  result :: low)
{
    if(pin == guess) {
        int b = balance();

        // declassify balance
        _(assume b :: low)
        return b;
    } else {
        return -1; // pin incorrect
    }
}
```

Declassification via `assume`

```
int get_balance(int pin, int guess)
    _(requires (pin == guess) :: low)
    _(requires guess :: low)
    _(ensures  result :: low)
{
    if(pin == guess) {
        int b = balance();

        // declassify balance
        _(assume b :: low)
        return b;
    } else {
        return -1; // pin incorrect
    }
}
```

Easy to add this proof rule:
Declassification by miracle.

$$\frac{}{\{P\}\ \texttt{assume}\ \ \rho\ \{P \land \rho\}}$$

Declassification via `assume`

```
int get_balance(int pin, int guess)
    _(requires (pin == guess) :: low)
    _(requires guess :: low)
    _(ensures  result :: low)
{
    if(pin == guess) {
        int b = balance();

        // declassify balance
        _(assume b :: low)
        return b;
    } else {
        return -1; // pin incorrect
    }
}
```

Easy to add this proof rule:
Declassification by miracle.

$$\frac{}{\{P\}\ \texttt{assume}\ \rho\ \{P \land \rho\}}$$

- ✓ intuitive & easy to use
- ✓ great *mechanism*
- ✗ dangerous (no crosscheck)

Declassification via `assume`

```
int get_balance(int pin, int guess)
    _(requires (pin == guess) :: low)
    _(requires guess :: low)
    _(ensures  result :: low)
{
    if(pin == guess) {
        int b = balance();

        // declassify balance
        _(assume b :: low)
        return b;
    } else {
        return -1; // pin incorrect
    }
}
```

Easy to add this proof rule:
Declassification by miracle.

$$\frac{}{\{P\}\ \texttt{assume}\ \ \rho\ \{P \land \rho\}}$$

- ✓ intuitive & easy to use
- ✓ great *mechanism*
- ✗ dangerous (no crosscheck)

Contribution: Semantic characterization of the soundness of this rule

Declassification via `assume`: what can go wrong

```
int get_balance(int pin, int guess)
    _(requires (pin == guess) :: low)
    _(requires guess :: low)
    _(ensures  result :: low)
{
    // insecure declassification
    _(assume pin :: low)
    if(pin == guess) {
        int b = balance();
        _(assume b :: low)
        return b;
    } else if (pin > guess) {
        return -1; // pin incorrect
    } else {
        return -2; // guess too low
    }
}
```

Declassification via `assume`: what can go wrong

```
int get_balance(int pin, int guess)
    _(requires (pin == guess) :: low)
    _(requires guess :: low)
    _(ensures  result :: low)
{
    // insecure declassification
    _(assume pin :: low)
    if(pin == guess) {
        int b = balance();
        _(assume b :: low)
        return b;
    } else if (pin > guess) {
        return -1; // pin incorrect
    } else {
        return -2; // guess too low
    }
}
```

Contribution: Justification of declassifications wrt. to high-level policies

$$\mathcal{D}(tr) = \phi(tr) \rightsquigarrow \rho(tr)$$

Declassification via `assume`: what can go wrong

```
int get_balance(int pin, int guess)
    _(requires (pin == guess) :: low)
    _(requires guess :: low)
    _(ensures  result :: low)
{
    // insecure declassification
    _(assume pin :: low)
    if(pin == guess) {
        int b = balance();
        _(assume b :: low)
        return b;
    } else if (pin > guess) {
        return -1; // pin incorrect
    } else {
        return -2; // guess too low
    }
}
```

Contribution: Justification of declassifications wrt. to high-level policies

$$\mathcal{D}(tr) = \phi(tr) \rightsquigarrow \rho(tr)$$

- tr: application-specific trace recording key events
- $\phi(tr)$: *when* declassification may occur given tr
- $\rho(tr)$: *what* information may be declassified subsequently

Declassification via `assume`: what can go wrong

```
int get_balance(int pin, int guess)
    _(requires (pin == guess) :: low)
    _(requires guess :: low)
    _(ensures  result :: low)
{
    // insecure declassification
    _(assume pin :: low)
    if(pin == guess) {
        int b = balance();
        _(assume b :: low)
        return b;
    } else if (pin > guess) {
        return -1; // pin incorrect
    } else {
        return -2; // guess too low
    }
}
```

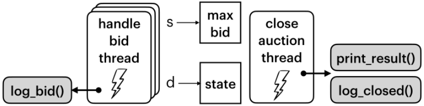
Contribution: Justification of declassifications wrt. to high-level policies

$$\mathcal{D}(tr) = \phi(tr) \rightsquigarrow \rho(tr)$$

- tr: application-specific trace recording key events
- $\phi(tr)$: *when* declassification may occur given tr
- $\rho(tr)$: *what* information may be declassified subsequently
- Demo!

Case-studies $\mathcal{D}(tr) = \phi(tr) \rightsquigarrow \rho(tr)$

Sealed-bid auction server

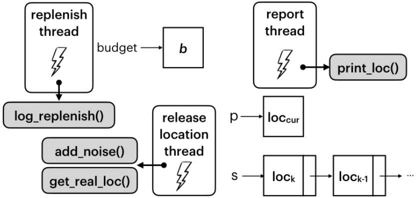


Policy

when: auction is finished

what: winning bid

Private location server



Policy

when: sufficient privacy budget

what: a noisy location point

. . . , Wordle, private learning, and more in the Git repository:
`src/master/examples/case-studies`

Goal: Correctness + Security Verification for C Code

- SecCSL: Security Concurrent Separation Logic [Ernst & Murray CAV 2019]
  - Integrates security reasoning into deductive verification
  - Allows semantic reasoning about information flow

- Assume but Verify [Murray+ CCS 2023]
  - Bridges between high-level policies and code-level proofs
  - Knowledge-based characterization of declassifications

- Project Overview: `https://covern.org/`
- Tool support & case studies: `https://bitbucket.org/covern/secc`
- Formalization and proofs are mechanized in Isabelle/HOL ✓

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

$$\frac{\{\varphi \land P\}\ c\ \{Q\} \quad \{\neg\varphi \land P\}\ c\ \{Q\}}{\{\varphi :: \texttt{low} \land P\}\ c\ \{Q\}}\ \textsc{Split}$$

- 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 $\textsc{Conseq}$)

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

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

Observable Locations

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

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
  - $\mathrm{lows}_\ell(P_1 \star \mathrm{invs}(L_1), s_1) \subseteq \mathrm{lows}_\ell(P_2 \star \mathrm{invs}(L_2), s_2)$