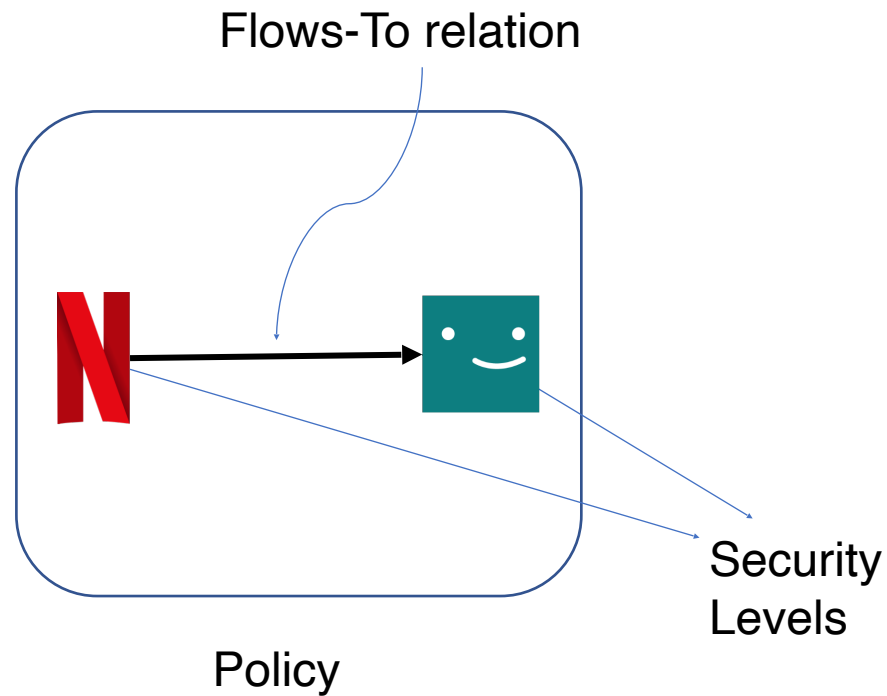


Design, implementation, and mechanisation of dynamic policies

Antonio Zegarelli, **Niki Vazou**, Marco Guarnieri
IMDEA Software Institute

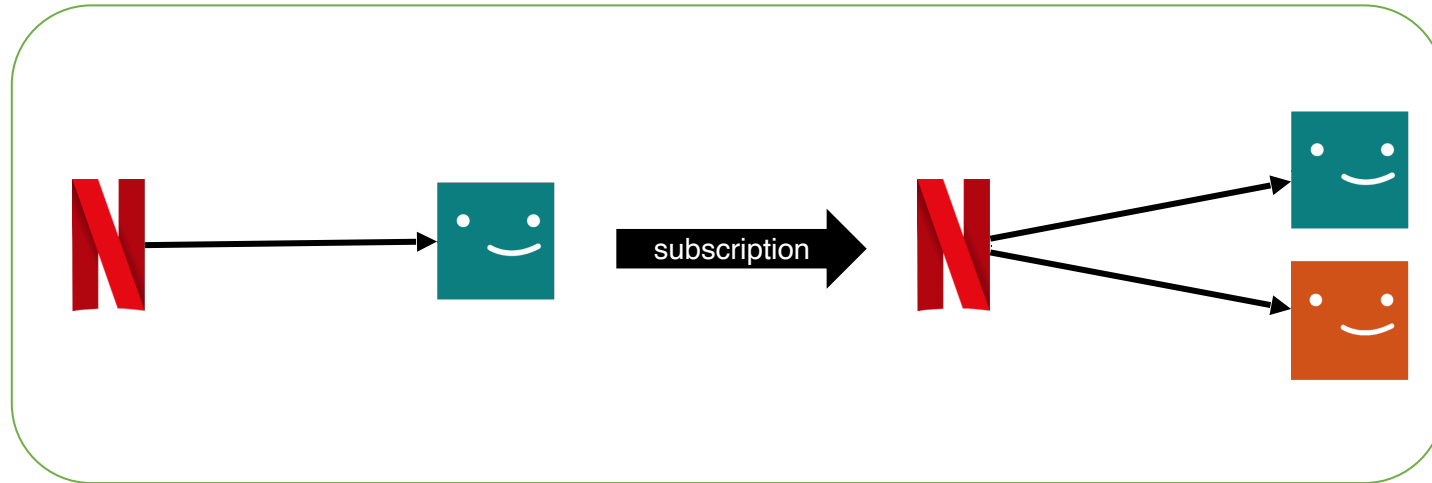
Information flow control

Secure exchange of information in a system



Information flow control

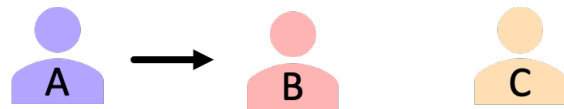
Secure exchange of information in a system



Dynamic Policy

Permitted flows can change during computation

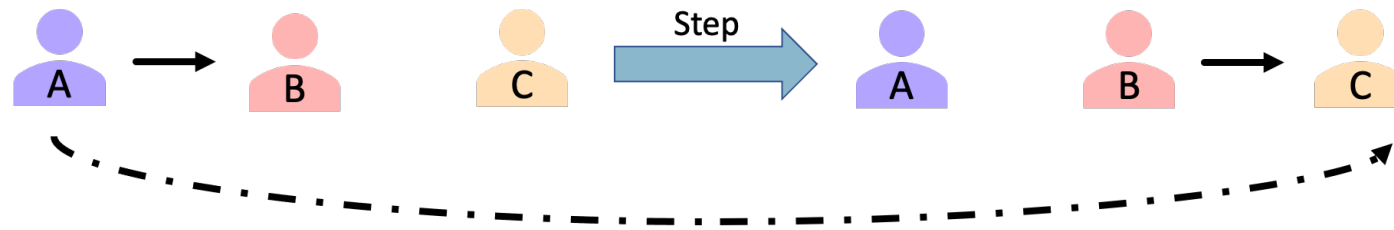
Example of Dynamic Policy



Example of Dynamic Policy

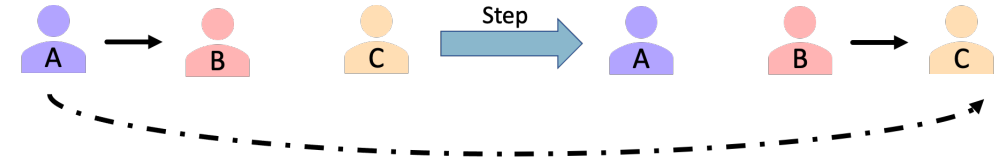


Is this safe?



**Information is shared from A to C via B
through time**

It depends!



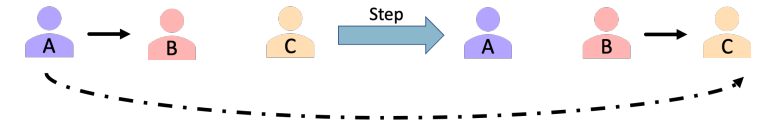
Allowed ✓

```
-- A = Input
-- B = Sanitizer
-- C = DB
Input ⊆ Sanitizer
s <- sanitize input
Input ⊈ Sanitizer
Sanitizer ⊆ DB
store s
```

Forbidden ✗

```
-- A = Patient
-- B = Hospital
-- C = Doc
Patient ⊆ Hospital
h <- receive patient
Patient ⊈ Hospital
Hospital ⊆ Doc
share h Doc -- CRASH
```

It depends!

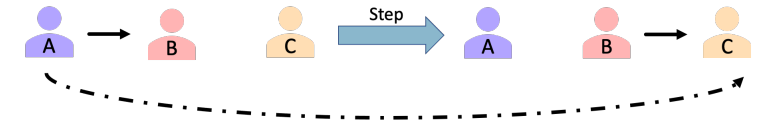


Allowed ✓

```
-- A = Input
-- B = Sanitizer
-- C = DB
Input  $\sqsubseteq$  Sanitizer
s  $\leftarrow$  sanitize input
Input  $\not\sqsubseteq$  Sanitizer
Sanitizer  $\sqsubseteq$  DB
store s
```

We want to **allow** the exchange

It depends!



Forbidden **X**

```
-- A = Patient
-- B = Hospital
-- C = Doc
Patient  $\sqsubseteq$  Hospital
h  $\leftarrow$  receive patient
Patient  $\not\sqsubseteq$  Hospital
Hospital  $\sqsubseteq$  Doc
share h Doc -- CRASH
```

We want to **forbid** the exchange

By knowing the value of h, DOC learns the patient's information

Safety of IF depends on the security policy.

There exist various interpretations of dynamic policies
(a.k.a. facets):



We just saw that one

- **Time-transitive**: moves information from A to C via B
- **Termination insensitive**: leak information via divergence
- **Replaying**: release of information is permanent
- **Whitelisting**: allowed by some other part of the policy

Design of Systems with Dynamic Policies

- ① **Security condition:** predicate that gives semantics to the policy to express the intended interpretation
- ② **Enforcement mechanism:** a monadic library that enforces the security condition
- ③ **Proof mechanisation:** that we can only build programs that satisfy the security condition

Security Condition

Can we securely execute a program e with an original state Σ ?

$$\langle \Sigma \mid e \rangle \Downarrow t \cdot \alpha$$

is an execution trace, i.e., there exists a sequence

$$\langle \Sigma \mid e \rangle \rightarrow \langle \Sigma_1 \mid e_1 \rangle \rightarrow \dots \rightarrow \langle \Sigma_n \mid e_n \rangle$$

To define the security condition for an attacker A , we use

Erasure $\varepsilon_A(\cdot)$ to encode observability and

Exclusion knowledge $ek_A(\cdot)$ to encode what is observed by an attacker.

Erasure

- Removes any non observable information
- Maps expressions to equivalence classes that cannot be distinguished by an attacker

```
-- A = Patient
-- B = Hospital
-- C = Doc
Patient  $\sqsubseteq$  Hospital
h  $\leftarrow$  receive patient
Patient  $\not\sqsubseteq$  Hospital
Hospital  $\sqsubseteq$  Doc
share h Doc -- CRASH
```

Erasure

- Removes any non observable information
- Maps expressions to equivalence classes that cannot be distinguished by an attacker

\mathcal{E}_{Doc}

```
-- A = Patient
-- B = Hospital
-- C = Doc
Patient  $\sqsubseteq$  Hospital
█  $\leftarrow$  receive █
Patient  $\not\sqsubseteq$  Hospital
Hospital  $\sqsubseteq$  Doc
share █ Doc -- CRASH $\swarrow$ 
```


Attacker's Knowledge

Exclusion Knowledge set

- All expressions that can be excluded since they can't produce the same observations

```
-- A = Patient
-- B = Hospital
-- C = Doc
Patient  $\sqsubseteq$  Hospital
█  $\leftarrow$  receive █
Patient  $\not\sqsubseteq$  Hospital
Hospital  $\sqsubseteq$  Doc
share █ Doc -- CRASH
```


Back to the example



```
-- A = Input
-- B = Sanitizer
-- C = DB
Input  $\sqsubseteq$  Sanitizer
█  $\leftarrow$  sanitize █
Input  $\not\sqsubseteq$  Sanitizer
Sanitizer  $\sqsubseteq$  DB
store █
```

$$ek_{DB} = \emptyset$$

Back to the example



```
-- A = Input  
-- B = Sanitizer  
-- C = DB  
Input  $\sqsubseteq$  Sanitizer  
s  $\leftarrow$  sanitize input  
Input  $\not\sqsubseteq$  Sanitizer  
Sanitizer  $\sqsubseteq$  DB  
store s
```

$$ek_{DB} = \{v \mid s \neq \text{sanitize } v\}$$

Exclusion Knowledge

The exclusion knowledge of an attacker A
i.e., the set of initial inputs that could not have lead to A 's observations.

$$ek_A(o) = \{e' \mid \neg \exists t'. \langle \Sigma \mid e' \rangle \Downarrow t' \text{ and } \varepsilon_A(t') = o\}$$

Why exclusion?

Because it is better to convey what information **cannot** be leaked

Exclusion Knowledge

The exclusion knowledge of an attacker A
i.e., the set of initial inputs that could not have lead to A 's observations.

$$ek_A(o) = \{e' \mid \neg \exists t'. \langle \Sigma \mid e' \rangle \Downarrow t' \text{ and } \varepsilon_A(t') = o\}$$

The exclusion progress knowledge of an attacker A
also requires that the final state is observable

$$ek_A^+(o) = \{e' \mid \neg \exists t', \alpha'. \langle \Sigma \mid e' \rangle \Downarrow t' \cdot \alpha' \text{ and } \varepsilon_A(t') = o \wedge obs_A(\alpha')\}$$

Security Condition

What the attacker A can exclude up to
the latest observation

What the attacker is allowed to
learn with the latest observation
according to the policy

$$\overline{ek_A(\varepsilon_A(t \cdot \alpha))} \setminus \underbrace{ek_A^+(\varepsilon_A(t))}_{\text{What the attacker } A \text{ can exclude knowing that the next step is observable}} \subseteq \overline{\mathcal{B}_A(t, \alpha)}$$


What the attacker A can exclude
knowing that the next step is
observable

where $\langle \Sigma \mid e \rangle \Downarrow t \cdot \alpha$

is an execution trace, i.e., there exists a sequence

$$\langle \Sigma \mid e \rangle \rightarrow \langle \Sigma_1 \mid e_1 \rangle \rightarrow \dots \rightarrow \langle \Sigma_n \mid e_n \rangle$$

Back to the example




```
-- A = Input
-- B = Sanitizer
-- C = DB
Input  $\sqsubseteq$  Sanitizer
█  $\leftarrow$  sanitize █
Input  $\not\sqsubseteq$  Sanitizer
Sanitizer  $\sqsubseteq$  DB
store █
```

$$ek_{DB} \setminus ek_{DB}^+ = \emptyset$$

$$B_{DB} = \emptyset$$

Back to the example




```
-- A = Input
-- B = Sanitizer
-- C = DB
Input  $\sqsubseteq$  Sanitizer
s  $\leftarrow$  sanitize input
Input  $\not\sqsubseteq$  Sanitizer
Sanitizer  $\sqsubseteq$  DB
store s
```

$$ek_{DB} \setminus ek_{DB}^+ = \{v \mid s \neq \text{sanitize } v\}$$

$$B_{DB} = \{v \mid s \neq \text{sanitize } v\}$$

Back to the example



```
-- A = Patient
-- B = Hospital
-- C = Doc
Patient  $\sqsubseteq$  Hospital
h  $\leftarrow$  receive patient
Patient  $\not\sqsubseteq$  Hospital
Hospital  $\sqsubseteq$  Doc
share h Doc -- CRASH $\nearrow$ 
```

$$ek_{Doc} \setminus ek_{Doc}^+ = \{v \mid s \neq \text{receive } v\}$$

$$B_{Doc} = \emptyset$$

Design of Systems with Dynamic Policies

- ① Security condition definition for each interpretation
- ② **Enforcement mechanism that satisfy the security condition**
- ③ Mechanization allowing for proof reuse

LIO: IFC in Haskell

```
class (Eq l) => Label l where  
  ( $\sqsubseteq$ ) :: l -> l -> Bool -- Can flow to  
  ( $\sqcup$ ) :: l -> l -> l -- Join  
  ( $\sqcap$ ) :: l -> l -> l -- Meet ()
```

```
L  $\sqsubseteq$  L = True  
L  $\sqsubseteq$  H = True  
H  $\sqsubseteq$  L = False  
H  $\sqsubseteq$  H = True
```

Labels are lattices, e.g., (L | H)
that are used to protect values

```
newtype Label l => Labeled l a = Lab l a
```

```
Lab H 42
```

LIO: IFC in Haskell

The labeled monad keeps track of the “current label”

```
newtype Label l  $\Rightarrow$  LIO l a = LIO (StateT l IO a)
```

```
 $\langle lc \mid e \rangle \rightarrow \langle lc \mid e \rangle$ 
```

To create values labeled l , the current label should flow to the l

```
label :: Label l  $\Rightarrow$  l  $\rightarrow$  a  $\rightarrow$  LIO l (Labeled l a)  
label l v = do  
  lc  $\leftarrow$  get  
  if lc  $\sqsubseteq$  l then return (Lab l v)  
  else throwError
```

```
 $\langle H \mid \text{label } L \ 42 \rangle$ 
```

```
 $\rightarrow$ 
```

```
 $\langle H \mid \text{Lab } L \ 42 \rangle$ 
```

```
 $\langle L \mid \text{label } H \ 42 \rangle$ 
```

```
 $\rightarrow$ 
```

```
 $\langle L \mid \text{error} \rangle$ 
```

To unlabel it raises the current label

```
unlabel :: Label l  $\Rightarrow$  Labeled l a  $\rightarrow$  LIO l a  
unlabel (Lab l v) = modify ( $\sqcup$  l) >> return v
```

```
 $\langle L \mid \text{unlabel (Lab } H \ 42) \rangle$ 
```

```
 $\rightarrow$ 
```

```
 $\langle H \mid 42 \rangle$ 
```

LIO: IFC in Haskell

LIO is very **simple**, has many extensions and applications, and comes **partially** verified in Coq.

But, does not support dynamic policies.

“Flexible Dynamic Information Flow Control in the presence of exceptions”, by Stefan, Russo, Mitchell, and Mazieres. JFP’17

“Addressing covert termination and timing channels in concurrent information flow systems”, by Stefan, Russo, Buiras, Levy, Mitchell, and Mazieres. ICFP’12. **Most Influential Paper**

SLIO: Stateful LIO for Dynamic Policies

The **state** now tracks the history of the flows-to relations

```
newtype SLIO l a = SLIO (StateT (l, [[(l,l)]] IO) a)
```

$$\langle lc, st \mid e \rangle \rightarrow \langle lc, st \mid e \rangle$$

Labelling now checks flows-to based on the state

```
label :: l → a → SLIO l (Labeled l a)
label l v = do
  (lc,st) ← get
  if lc  $\sqsubseteq_{st}$  l then return (Lab l v)
  else throwError
```

$$lc \sqsubseteq_{st} l = (lc,l) \in \text{closure } st$$

SLIO: Stateful LIO for Dynamic Policies

SLIO was the first LIO extension that supports dynamic policies and comes with a paper and pencil security proof.

But, it only captures time-transitive flows.

DynLIO: Dynamic Policies

To capture VARIOUS dynamic policies we “just” make the label guard parametric

```
label :: l → a → SLIO l (Labeled l a)
label l v = do
  (lc, st) ← get
  if lc ⊑st l then return (Lab l v)
  else throwError
```

```
class IFC l st where
  guard :: l → st → Bool
```

DynLIO: Dynamic Policies

To capture VARIOUS dynamic policies we “just” make the label guard parametric

```
label :: IFC l st  $\Rightarrow$  l  $\rightarrow$  a  $\rightarrow$  DynLIO l st (Labeled l a)
label l v = do
  st  $\leftarrow$  get
  if guard l st then return (Lab l v)
  else throwError
```

```
class IFC l st where
  guard :: l  $\rightarrow$  st  $\rightarrow$  Bool
```

```
 $\langle i, st \mid e \rangle \rightarrow \langle st \mid e \rangle$ 
```

“Just” because in reality IFC has more methods for other IFC functions, e.g., toLabeled

DynLIO: Dynamic Policies

```
class IFC l st where  
  guard :: l → st → Bool
```

No Time Transitive

```
instance IFC l _ where  
  guard l (lc,st) = (lc, l) ∈ closure st
```

Time Transitive

```
instance IFC l _ where  
  guard l (lc,st) = (lc, l) ∈ latest st
```

Simple LIO

```
instance Label l ⇒ IFC l l where  
  guard l lc = lc ⊆ l
```


DynLIO for Dynamic Policies

DynLIO provides a modular interface to encode **various** dynamic policies.

It is complicated to define a new instance (e.g., `toLabeled` definition is omitted and requires some complicated state) but easy to use.

We have used it to define instances for LIO, no time transitive, time transitive, and replaying.

We hope it can also be used for a modular proof.

Design of Systems with Dynamic Policies

- ① Security condition definition for each interpretation
- ② Enforcement mechanism that satisfy the security condition
- ③ **Mechanization aiming for proof reuse**

Mechanisation of IFC is difficult

LIO's incomplete mechanisation had an error
(on the definition of erasure).

SLIO's paper and pencil proof had an error
(on the definition of erasure).

There is no mechanised proof of dynamic policies.

Mechanisation of DynLIO in Liquid Haskell

Expressions are embedded as a data type

```
data Expr l
  = ELabel (Expr l) (Expr l)
  | ELab l | ERet (Expr l) ...
```

Evaluation is defined parametrised with the IFC methods

```
data Step l st where
  SLabel :: i:IFC l st → st:State l → l:{eGuard i st l} → e:Expr l
    → Step i (Pg st (ELabel (ELab l) e))
      (Pg st (ERet (ELB (ELab l) e))
```

Mechanisation of DynLIO in Liquid Haskell

The security condition is encoded into a refinement type

$$ek_A(\varepsilon_A(t \cdot \alpha)) \setminus ek_A^+(\varepsilon_A(t)) \subseteq \mathcal{B}_A(t, \alpha)$$

```
security :: l:Label → s_0:State
  → e_1:Expr l → e_2:Expr l

  → t_n:Trace l st → s_n:State
  → e_n:ObsEvalExpr l s_0 e_1 t_n s_n → s:{State | st s_n = s }

  → t_m:Trace l st → s_m:State
  → e_m:ObsEvalExpr l s_0 e_2 t_m s_m

  → LowEquivTrace l t_n t_m
  → InputRelease l s e_1 e_2
  → SecurityBound l st t_n t_m
  → LowEquivTrace l (t_n >> (Pg s_n e_n)) (t_m >> (Pg s_m e_m))
```

Goal:
Define security

Mechanisation of DynLIO in Liquid Haskell

Currently:

security is almost defined for the time transitive instance
all the development (with comments) is 10K lines of code
security is property base tested
(to be presented at Lambda Days by Fernanda Andrade)

Next:

Define security for more instances and hope to observe proof reuse
Compose instances to define new dynamic policies

Design of Systems with Dynamic Policies

- ① **Security condition:** predicate that gives semantics to the policy to express the intended interpretation
- ② **Enforcement mechanism:** dynLIO, a monadic and parametric library that enforces the security condition
- ③ **Proof mechanisation:** proof the dynLIO actually enforces the security condition under development, with aim of modularity

Thanks!