Poison-pills and dynamic information flow control

Cătălin Hrițcu
(joint work with Michael Greenberg, Benoît Montagu, Greg Morrisett, Benjamin Pierce, Randy Pollack, …)

Penn PLClub - 2012-05-11

Outline

- Dynamic information flow control (IFC)
- Poison-pill attacks for dynamic IFC (informally)
- Solution ingredients:
  - Public labels
  - No fatal errors
- Defining poison-pill attacks and protection

Purely dynamic IFC

λ_V

$$\rho, pc \vdash e \Downarrow v @ l$$

Purely dynamic IFC

$\boxed{\lambda_V}$

$$\boxed{\rho, pc \vdash e \Downarrow v@l}$$

$$\frac{\rho, pc \vdash e \Downarrow b@lb \quad b \in \{\mathsf{true}, \mathsf{false}\} \qquad \rho, pc \vee lb \vdash e_b \Downarrow v@l}{\rho, pc \vdash \mathsf{if}\ e\ \mathsf{then}\ e_{\mathsf{true}}\ \mathsf{else}\ e_{\mathsf{false}} \Downarrow v@l}$$

pc prevents implicit flows:

```
let rpub = ref public () in
if bit@secret then rpub := true
              else rpub := false
```

Purely dynamic IFC

$\boxed{\lambda_V}$

$$\boxed{\rho, pc \vdash e \Downarrow v@l}$$

$$\frac{\rho, pc \vdash e \Downarrow b@lb \quad b \in \{\mathsf{true}, \mathsf{false}\} \qquad \rho, pc \vee lb \vdash e_b \Downarrow v@l}{\rho, pc \vdash \mathsf{if}\ e\ \mathsf{then}\ e_{\mathsf{true}}\ \mathsf{else}\ e_{\mathsf{false}} \Downarrow v@l}$$

$$\frac{\rho(x) = v@l}{\rho, pc \vdash x \Downarrow v@(l \vee pc)}$$

$$\frac{}{\rho, pc \vdash \lambda x.e \Downarrow \langle \rho, \lambda x.e \rangle @pc}$$

pc “infects” all values created on high branch
(we need this because of automatic pc restoration)

```
let rpub = ref public () in
let copy = (if bit@secret then true
                           else false)
in rpub := copy // pc restored, leak secret?
```

Purely dynamic IFC

λ_V

$$\boxed{\rho, pc \vdash e \Downarrow v @ l}$$

$$\frac{\rho, pc \vdash e \Downarrow b @ lb \quad b \in \{\mathsf{true}, \mathsf{false}\} \qquad \rho, pc \vee lb \vdash e_b \Downarrow v @ l}{\rho, pc \vdash \mathsf{if}\ e\ \mathsf{then}\ e_{\mathsf{true}}\ \mathsf{else}\ e_{\mathsf{false}} \Downarrow v @ l}$$

$$\frac{\rho(x) = v @ l}{\rho, pc \vdash x \Downarrow v @ (l \vee pc)}$$

$$\frac{}{\rho, pc \vdash \lambda x.e \Downarrow \langle \rho, \lambda x.e \rangle @ pc}$$

$$\frac{\rho, pc \vdash e_1 \Downarrow \langle \rho', \lambda x.e \rangle @ l_1 \qquad \rho, pc \vdash e_2 \Downarrow v_2 @ l_2 \qquad \rho'[x \mapsto v_2 @ l_2], pc \vee l_1 \vdash e \Downarrow v_3 @ l_3}{\rho, pc \vdash e_1\ e_2 \Downarrow v_3 @ l_3}$$

Purely dynamic IFC

λ_V

$$\rho, pc \vdash e \Downarrow v@l$$

$$\frac{\rho, pc \vdash e \Downarrow b@lb \quad b \in \{\mathsf{true}, \mathsf{false}\} \qquad \rho, pc \vee lb \vdash e_b \Downarrow v@l}{\rho, pc \vdash \mathsf{if}\ e\ \mathsf{then}\ e_{\mathsf{true}}\ \mathsf{else}\ e_{\mathsf{false}} \Downarrow v@l}$$

$$\frac{\rho(x) = v@l}{\rho, pc \vdash x \Downarrow v@(l \vee pc)}$$

$$\frac{}{\rho, pc \vdash \lambda x.e \Downarrow \langle \rho, \lambda x.e \rangle @pc}$$

$$\frac{\rho, pc \vdash e_1 \Downarrow \langle \rho', \lambda x.e \rangle @l_1 \qquad \rho, pc \vdash e_2 \Downarrow v_2@l_2 \qquad \rho'[x \mapsto v_2@l_2], pc \vee l_1 \vdash e \Downarrow v_3@l_3}{\rho, pc \vdash e_1\ e_2 \Downarrow v_3@l_3}$$

Non-interference

(termination & error insensitive)

$$\left.\begin{array}{l} \rho_1, pc \vdash e \Downarrow v_1@l_1 \\ \rho_2, pc \vdash e \Downarrow v_2@l_2 \\ \rho_1 \simeq_l \rho_2 \end{array}\right\} \Rightarrow v_1@l_1 \simeq_l v_2@l_2$$

[Austin & Flanagan, PLAS 2009] [Breeze Summer 2011]

Mutable state

$$\boxed{\rho, pc \vdash e, \sigma \Downarrow v@l, \sigma'}$$

very easy with **weak updates**

(references have fixed labels set on creation time)

Mutable state

$$\boxed{\rho, pc \vdash e, \sigma \Downarrow v@l, \sigma'}$$

very easy with **weak updates**

(references have fixed labels set on creation time)

$$\frac{\rho(x) = v@l_v \quad r \notin \sigma \quad l_v \vee pc \sqsubseteq l_r}{\rho, pc \vdash \texttt{ref}\ l_r\ x, \sigma \Downarrow r@pc, \sigma[r \mapsto v@l_r]}$$

$$\frac{\rho(x) = v@l_v \quad \sigma(r) = v'@l_r \quad l_v \vee pc \sqsubseteq l_r}{\rho, pc \vdash r := x, \sigma \Downarrow r@pc, \sigma[r \mapsto v@l_r]}$$

$$\frac{\sigma(r) = v@l_r}{\rho, pc \vdash !r, \sigma \Downarrow v@(l_r \vee pc), \sigma}$$

Mutable state

$$\boxed{\rho, pc \vdash e, \sigma \Downarrow v@l, \sigma'}$$

very easy with **weak updates**

(references have fixed labels set on creation time)

$$\frac{\rho(x) = v@l_v \quad r \notin \sigma \quad l_v \vee pc \sqsubseteq l_r}{\rho, pc \vdash \mathsf{ref}\ l_r\ x, \sigma \Downarrow r@pc, \sigma[r \mapsto v@l_r]}$$

$$\frac{\rho(x) = v@l_v \quad \sigma(r) = v'@l_r \quad l_v \vee pc \sqsubseteq l_r}{\rho, pc \vdash r := x, \sigma \Downarrow r@pc, \sigma[r \mapsto v@l_r]}$$

$$\frac{\sigma(r) = v@l_r}{\rho, pc \vdash !r, \sigma \Downarrow v@(l_r \vee pc), \sigma}$$

$$\frac{\rho, pc \vdash e_1, \sigma \Downarrow v_1@l_1, \sigma' \qquad \rho[x \mapsto v_1@l_1], pc \vdash e_2, \sigma' \Downarrow v_2@l_2, \sigma''}{\rho, pc \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2, \sigma \Downarrow v_2@l_2, \sigma''}$$

[Breeze Summer 2011]

Simplest DynIFC poison-pill attack

- ```
  let log = ref public 0
  fun server x y =
    let res = x + y in
    log := !log + res;
    res
  ```
  ```

Simplest DynIFC poison-pill attack

- ```
  let log = ref public 0
  fun server x y =
    let res = x + y in
    log := !log + res;
    res
  ```
  server expects public numbers
  ```

Simplest DynIFC poison-pill attack

- ```
  let log = ref public 0
  fun server x y =
    let res = x + y in
    log := !log + res;
    res
  ```
  server expects public numbers

- ```
  let attacker =
    server 1 (2@secret)
  ```
  attacker sends secret pill

Simplest DynIFC poison-pill attack

- `let log = ref public 0` — server expects public numbers
  ```
  fun server x y =
    let res = x + y in
    log := !log + res;
    res
  ```
  `res=3@High`
  attempted write down to `log`
  server gets killed (fatal IFC error)
  -> availability attack
- ```
  let attacker =
    server 1 (2@secret)
  ```
  attacker sends secret pill
  ```

Simplest DynIFC poison-pill attack

- `let log = ref public 0` — server expects public numbers

  ```
  fun server x y =
    let res = x + y in
    log := !log + res;
    res
  ```

  `res=3@High`

  attempted write down to `log`

  server gets killed (fatal IFC error)

  -> availability attack

- `let attacker =`
  `server 1 (2@secret)` — attacker sends secret pill

Poison-pill problem:

in λ_V we can't protect this server against poison-pills

Trying to protect server

- ```
  let log = ref public 0
  fun server x y =
    if labelOf x == public &&
       labelOf y == public then
      let res = x + y in
      log := !log + res;
      res
    else "pls stop poison"
  ```
  ```

Trying to protect server

- ```
  let log = ref public 0
  fun server x y =
    if labelOf x == public &&
       labelOf y == public then
      let res = x + y in
      log := !log + res;
      res
    else "pls stop poison"
  ```

- We need **public labels** for this:

$$\frac{\rho(x) = v@l}{\rho, pc \vdash \mathsf{labelOf}\ x, \sigma \Downarrow l@\bot, \sigma}$$
  ```

Problem: labelOf unsound in λ_V

- Labels themselves are an IF channel
  - ```
    let x = (if bit@secret then ()@secret
                           else ()@topSecret)
    in copy = (labelOf x == secret)
    ```
- Public labels unsound if pc restored automatically

$$\frac{\rho(x) = v@l}{\rho, pc \vdash \mathsf{labelOf}\ x, \sigma \Downarrow l@\bot, \sigma}$$
    ```

Problem: labelOf unsound in λ_V

- Labels themselves are an IF channel
  - ```
    let x = (if bit@secret then ()@secret
    declPc public;              else ()@topSecret)
    in copy = (labelOf x == secret)
    ```
- Public labels unsound if pc restored automatically
- Manual pc declassification doesn't work
  - Adds many subtle audit points … mostly spurious

[Safe-breeze] Manual PC declassification considered harmful

6 messages

Benjamin C. Pierce <bcpierce@cis.upenn.edu> **Wed, Aug 17, 2011 at 1:24 PM**
To: safe-breeze@lists.crash-safe.org

A few months ago, we made the decision that it was better to remove the "automatic declassification of the PC" at the ends of conditionals and functions in Breeze (and at return instructions in the ISA) and, instead, demand that programmers lower the PC manually, if it becomes higher than they want it. Over the past few days, we've finally made this change to Breeze and have been experimenting with programming in this style. Our conclusion, sadly, is that it doesn't work.
    ```

Solution: brackets

λ_{WP}

- Manual pc restoring
  - ```
    let x = topSecret<if bit@secret then ()@secret
                                    else ()@topSecret>
    in copy = (labelOf x == secret)
    ```
    ```

Solution: brackets

λ_{WP}

- Manual pc restoring
  - ```
    let x = topSecret<if bit@secret then ()@secret
                                     else ()@topSecret>
    in copy = (labelOf x == secret)
    ```

$$\frac{\rho \vdash e, pc \Downarrow v @ l, pc' \quad l \vee pc' \sqsubseteq lb \vee pc}{\rho \vdash lb \langle e \rangle , pc \Downarrow v @ lb, pc}$$
    ```

Solution: brackets

λ_{WP}

- Manual pc restoring
  - ```
    let x = topSecret<if bit@secret then ()@secret
                                     else ()@topSecret>
    in copy = (labelOf x == secret)
    ```

$$\frac{\rho \vdash e, pc \Downarrow v@l, pc' \quad l \vee pc' \sqsubseteq lb \vee pc}{\rho \vdash lb\langle e \rangle, pc \Downarrow v@lb, pc}$$

- Final label cannot depend on secrets (copy is always false)
- Programmer must predict pc & result label at the end of all branches
- Not a declassification construct
- pc no longer infectious (but result value still protected by pc)
- Without automatic pc restoration labels can be made public

[HAILS 2011, Breeze Fall 2011]
    ```

Poison-pill vulnerable server2

- ```
  let log = ref public 0
  fun server2 xs =
    let res = fold (+) xs 0 in
    log := !log + res;
    res
  ```

- `let attacker1 = server2 [1,2,42@secret]`
- `let attacker2 = server2 [1,2,42]@secret`
- `let attacker3 = server2 (1 :: [2,42]@secret)`
  ```

One way to protect server2

- ```
  let log = ref public 0
  fun server2 xs =
    rec fun sum xs =
      if labelOf xs == public then
        case xs of
          Cons x xs' =>
            if labelOf x == public then
              case sum xs' of
                Some s => Some (x + s)
                None => None
            else None
          Nil => Some 0
      else None
    case sum xs of
      Some res =>
        log := !log + res;
        res
      None => "error"
  ```
  ```

Wishful way to protect server2

- ```
  let log = ref public 0
  fun server2 xs =
    try
      public<
        let res = fold (+) xs 0 in
        log := !log + res;
        res
      >
    catch _ => "error"
  ```

Exceptions instead of fatal errors?

(the only way wrapping could work is if no error is fatal)
  ```

More reasons for exceptions

- “Stop the world” errors completely unrealistic
  - trying to build DynIFC-enforcing HW and an OS
  - shut down only the offending thread?
    - who gets to find out about the failure? does that leak?
    - does the thread get restarted? does that leak more?

More reasons for exceptions

- "Stop the world" errors completely unrealistic
  - trying to build DynIFC-enforcing HW and an OS
  - shut down only the offending thread?
    - who gets to find out about the failure? does that leak?
    - does the thread get restarted? does that leak more?
- Error insensitive non-interference very weak
  - security guarantees depend on fatality of errors

$$\left.\begin{array}{l} \rho_1, pc \vdash e \Downarrow v_1@l_1 \\ \rho_2, pc \vdash e \Downarrow v_2@l_2 \\ \rho_1 \simeq_l \rho_2 \end{array}\right\} \Rightarrow v_1@l_1 \simeq_l v_2@l_2$$

Exceptions vs. DynIFC

- Exceptions can be used to leak secrets
  - ```
    let rpub = ref public ()
    try
      secret<
        if bit@secret then throw E else ()
      >;
      rpub := false;
    catch E => rpub := true
    ```
- Exceptions destroy "decent" control flow
  - DynIFC relies on this for restoring the pc
    ```

The high price of exceptions

- Raise pc on operations that can cause exceptions
  - in Breeze all operations can cause exceptions
  - consequence: more brackets = annotations or/and more declassifications = audit points

The high price of exceptions

- Raise pc on operations that can cause exceptions
  - in Breeze all operations can cause exceptions
  - consequence: more brackets = annotations or/and more declassifications = audit points
- Try-catch cannot restore pc
  - not guaranteed control-flow merge point

The high price of exceptions

- Raise pc on operations that can cause exceptions
  - in Breeze all operations can cause exceptions
  - consequence: more brackets = annotations or/and more declassifications = audit points
- Try-catch cannot restore pc
  - not guaranteed control-flow merge point
- Brackets have to catch all exceptions
  - brackets could return labeled option
  - can choose when to handle (raise pc again)

The high price of exceptions

- Raise pc on operations that can cause exceptions
  - in Breeze all operations can cause exceptions
  - consequence: more brackets = annotations or/and more declassifications = audit points
- Try-catch cannot restore pc
  - not guaranteed control-flow merge point
- Brackets have to catch all exceptions
  - brackets could return labeled option
  - can choose when to handle (raise pc again)
- Two kinds of exceptions: active + delayed

TOO

The high price of exceptions

λ_{BP}

- Raise pc on operations that can cause exceptions
  - in Breeze all operations can cause exceptions
  - consequence: more brackets = annotations or/and more declassifications = audit points
- Try-catch cannot restore pc
  - not guaranteed control-flow merge point
- Brackets have to catch all exceptions
  - brackets could return labeled option
  - can choose when to handle (raise pc again)
- Two kinds of exceptions: active + delayed

But it does have error-sensitive non-interf.

Not-A-Value (NAV)

- Lower cost exception handling mechanism
- Idea: use only delayed exceptions (lazy)
- All values are morally labeled options / sums
  - [Tony Hoare, Null References: The Billion Dollar Mistake, 1965/2009]
- Exception propagation via data flow
  - no additional control flow
  - pc doesn't raise more
  - no bad interaction with brackets

Protecting server2 with NAVs

```
let log = ref public 0
fun server2 xs =
  let ores = public<fold (+) xs 0> in
  case ores of
    val res =>
      bind _ <- (log := !log + res) in
      res
    nav E => E
```

DEFINING POISON-PILLS

Not just DynIFC poison-pills

- type error pps (dynamic typing)
- contract failure pps (dynamic contracts)
- access control pps (IF-based access control)
- zero-order vs. higher-order pps
  - non-termination pps
  - resource consumption pps
- fast-acting (types*, contracts*, access*)
  vs. slow-acting pps (IFC, termination, resources)

*assuming fatal errors / eager exceptions

White-box vs. black-box protection

- White-box = rewriting
  - weaker protection; bigger overhead
  - not clear how to handle higher-order pps
    - does rewriting need to happen at run-time?
  - not having at least this means broken language
- Black-box = wrapping
  - stronger protection; smaller overhead
  - easier to handle higher-order pps
  - needs more mechanism, e.g. exception handling

Poison-pill protection

Definition (White-box protection against higher-order $\mathcal{B}$-poison-pills wrt $\mathcal{E}$)**.** *A deterministic language $\mathscr{L}$ provides white-box protection against higher-order $\mathcal{B}$-poison-pills with respect to $\mathcal{E}$ iff there exists a computable compositional transformation function $[\![\cdot]\!]$ from terms to terms, such that for every closed term t and every initial partial configuration $\mathscr{C}$:*

- *If $\mathscr{C}[t] \longrightarrow^{\star} \mathscr{C}' \nrightarrow$ then $\mathscr{C}[[\![t]\!]] \longrightarrow^{\star} \mathscr{C}'' \nrightarrow$ and $\neg\mathcal{B}(\mathscr{C}'')$ and additionally if $\neg\mathcal{B}(\mathscr{C}')$ then also $(\mathscr{C}', \mathscr{C}'') \in \mathcal{E}$.*

Poison-pill protection

Definition (White-box protection against higher-order $\mathcal{B}$-poison-pills wrt $\mathcal{E}$)**.** *A deterministic language $\mathscr{L}$ provides white-box protection against higher-order $\mathcal{B}$-poison-pills with respect to $\mathcal{E}$ iff there exists a computable compositional transformation function $[\![\cdot]\!]$ from terms to terms, such that for every closed term t and every initial partial configuration $\mathscr{C}$:*

- *If $\mathscr{C}[t] \longrightarrow^\star \mathscr{C}' \nrightarrow$ then $\mathscr{C}[[\![t]\!]] \longrightarrow^\star \mathscr{C}'' \nrightarrow$ and $\neg\mathcal{B}(\mathscr{C}'')$ and additionally if $\neg\mathcal{B}(\mathscr{C}')$ then also $(\mathscr{C}', \mathscr{C}'') \in \mathcal{E}$.*

Definition (White-box protection against **zero-order** $\mathcal{B}$-poison-pills wrt $\mathcal{E}$)**.** *…such that for every closed zero-order term t and any context C and any initial partial configuration $\mathscr{C}$:*

- *if $\mathscr{C}[C[t]] \longrightarrow^\star \mathscr{C}' \nrightarrow$ then $\mathscr{C}[[\![C[t]]\!]] \longrightarrow^\star \mathscr{C}'' \nrightarrow$ and $\neg\mathcal{B}(\mathscr{C}'')$ and additionally if $\neg\mathcal{B}(\mathscr{C}')$ then also $(\mathscr{C}', \mathscr{C}'') \in \mathcal{E}$.*

Poison-pill protection

Definition (White-box protection against higher-order $\mathcal{B}$-poison-pills wrt $\mathcal{E}$). *A deterministic language $\mathscr{L}$ provides white-box protection against higher-order $\mathcal{B}$-poison-pills with respect to $\mathcal{E}$ iff there exists a computable compositional transformation function $[\![\cdot]\!]$ from terms to terms, such that for every closed term t and every initial partial configuration $\mathscr{C}$:*

- *If $\mathscr{C}[t] \longrightarrow^{\star} \mathscr{C}' \nrightarrow$ then $\mathscr{C}[[\![t]\!]] \longrightarrow^{\star} \mathscr{C}'' \nrightarrow$ and $\neg\mathcal{B}(\mathscr{C}'')$ and additionally if $\neg\mathcal{B}(\mathscr{C}')$ then also $(\mathscr{C}', \mathscr{C}'') \in \mathcal{E}$.*

Definition (**Black-box** protection against higher-order $\mathcal{B}$-poison-pills wrt $\mathcal{E}$). *… iff there exists a context C_U so that for every closed term t for a fixed transformation $[\![t]\!] \triangleq C_U[t]$*

- *If $\mathscr{C}_U[t] \longrightarrow^{\star} \mathscr{C}'_U \nrightarrow$ then $\mathscr{C}_U[[\![t]\!]] \longrightarrow^{\star} \mathscr{C}'' \nrightarrow$ and $\neg\mathcal{B}(\mathscr{C}'')$ and additionally if $\neg\mathcal{B}(\mathscr{C}')$ then also $(\mathscr{C}', \mathscr{C}'') \in \mathcal{E}$.*

Plan / open questions

λ_V purely functional DynIFC language
prove it does not provide white-box protection

λ_{WP} public labels + brackets
prove white-box protection + no black-box protection

Q: Where do NAVs fit?

λ_{BP} exceptions
prove black-box protection

Plan / open questions

λ_V purely functional DynIFC language
prove it does not provide white-box protection

λ_{WP} public labels + brackets
prove white-box protection + no black-box protection

Q: Where do NAVs fit?

λ_{BP} exceptions
prove black-box protection

Q
Reasonable definitions? General enough?
Prove metaproperties about definitions?
Does any of this extend to state? concurrency?
Does this work in practice (Breeze)? ...

BACKUP SLIDES

DynIFC vs reliability

- DynIFC is a source of errors/exceptions
- DynIFC is a source of restrictions on reporting and handling errors/exceptions
  - exceptions are themselves a channel
  - e.g. Asbestos does very strange stuff like silently hiding errors

Program context (pc)

- pc prevents implicit flows

```
let rpub = ref public () in
if bit@secret then rpub := true
              else rpub := false
```

- pc "infects" all values created on high branch

$$\rho, pc \vdash e, \sigma \Downarrow v@l, \sigma' \Rightarrow l \sqsubseteq pc$$

  – we need this because of **automatic pc restoration**

```
let rpub = ref public () in
let copy = (if bit@secret then true
                           else false)
in rpub := copy // pc restored, leak secret?
```

Another solution for attack 1

```
let log = ref public 0
fun server xs =
  rec fun check_input xs =
    if labelOf xs == public then
      case xs of
        Cons x xs' =>
           (labelOf x == public) && f xs'
        Nil => true
    else false
  if check_input xs then
    let res = fold (+) xs 0 in
    log := !log + res;
    res
  else "error"
```

Poison-pill attack 2

- ```
  let log = ref secret []
  fun server2 xs =
    let res = fold (\x.\s. log:=x::!log; x+s)
                   xs 0
    in res
  ```

- ```
  let attacker2 =
    server2 [1,2,42@topSecret]
  ```

Poison-pill attack 3

- ```
  let pLog = ref public 0
  let sLog = ref secret 0
  fun server xs =
    let res = fold (+) xs 0 in
    sLog := !sLog + res;    // <- this fails
    pLog := !pLog + 1;
    res
  ```

- ```
  let attacker =
    server [1,2,42@topSecret]@secret
  ```

Non-solution for attack 3

- ```
  let pLog = ref public 0
  let sLog = ref secret 0
  fun server xs =
    if labelOf xs <: secret &&
       forall (\x. labelOf x <: secret) xs then
      let res = fold (+) xs 0 in
      sLog := !sLog + res;
      pLog := pLog + 1; // <- pc too high
      res
  ```
  ```

Better solution for attack 3

- ```
  let pLog = ref public 0 // counts total requests
  let sLog = ref secret 0 // only successful operations
  fun server xs =
    if labelOf xs <: secret then
      let ores =
        secret<fold (\x.\os.
          case os of
            Some s => if labelOf x == secret then Some(x+s)
                      else None
            None => None
          ) xs 0> in
      secret<case ores of
               Some res => sLog := !sLog + res;
               None => ()>
      pLog := !pLog + 1;
      secret<case ores of
               Some res => res;
               None => "pls stop poison">
  ```
  ```

A simpler + smarter solution for 3

```
• let pLog = ref public 0
  let sLog = ref secret 0
  fun server xs =
    if labelOf xs <: secret then
      let valid = secret<
        forall (\x. labelOf x <: secret) xs> in
      let res = secret<
        if valid then fold (+) xs 0 else 0
      > in
      sLog := !sLog + res;
      pLog := pLog + 1;
      res
```

Poison-pill attack ingredients

- Dynamic IFC
- Fine-grained labeling
  - High data can be hidden under low labels [1,2,pill@H]@L
- Decentralized label model
  - any code can classify data only for itself
- Fatal errors