Security II - Declassification

Stefano Calzavara

Università Ca' Foscari Venezia

April 16, 2020

Introduction

A secret is something that is told to one person at a time...

NI does not allow us to tell secrets to anyone, but sometimes we want to reveal some information which depends on a secret

- PIN checking: failing a login does reveal something about the PIN
- statistical information: aggregating information from multiple users, e.g., computing the average, might be an acceptable privacy loss

In this lecture we deal with declassification, i.e., the process of downgrading confidential information to public data.

Intentional Information Leaks

```
if (h == 4321) {
  l := 1;
  print("Welcome!");
}
else {
  l := 0;
  print("Error!");
}
```

Before execution:

$$\{h \mapsto 4321, l \mapsto 0\} \approx_L \{h \mapsto 5678, l \mapsto 0\}$$

After execution:

$$\{h \mapsto 4321, l \mapsto 1\} \not\approx_L \{h \mapsto 5678, l \mapsto 0\}$$

PIN checking violates non-interference!

Declassification

New syntactic construct to declassify confidential information to level ℓ

$$e ::= v \mid x \mid e + e \mid \ldots \mid \textbf{declassify}(e, \ell)$$

No semantic import: $\textbf{declassify}(e, \ell)$ is equivalent to e, it is just useful to explicitly annotate where declassification happens.

Programs with declassification violate NI: which security properties hold in presence of declassification?

Example: Average Salary

The following program violates non-interference:

$$l := (h_1 + h_2 + h_3)/3$$

If we consider this leak of information acceptable, we can rewrite the program as follows:

$$l := \textbf{declassify}((h_1 + h_2 + h_3)/3, L)$$

Nothing too interesting so far...

Laundering Attack

What about this program?

$$h_2 := h_1; h_3 := h_1; l := \textbf{declassify}((h_1 + h_2 + h_3)/3, L)$$

We are now in troubles! Why should we reject this one and accept the original one we had?

Key difference of the two programs: what is actually declassified!

Delimited Release

Suppose the program c contains exactly n declassify expressions:

$$\textbf{declassify}(e_1, \ell_1), \ldots, \textbf{declassify}(e_n, \ell_n).$$

Program c is secure w.r.t delimited release if and only if for all labels ℓ and memories μ_1, μ_2 such that:

1. $\mu_1 \approx_\ell \mu_2$
2. for all $\ell_i \sqsubseteq \ell$, $\langle e_i, \mu_1 \rangle$ and $\langle e_i, \mu_2 \rangle$ evaluate to the same value v_i

we have: if $\langle c, \mu_1 \rangle \Downarrow \mu_1'$ and $\langle c, \mu_2 \rangle \Downarrow \mu_2'$, then $\mu_1' \approx_\ell \mu_2'$

Delimited Release and Average Salary

Delimited release rejects the previous laundering attack:

$$h_2 := h_1; h_3 := h_1; l := \mathbf{declassify}((h_1 + h_2 + h_3)/3, L)$$

In particular, we can pick $\mu_1 \approx_L \mu_2$ as follows:

$$\begin{array}{rcl} \mu_1 & = & \{h_1 \mapsto 2, h_2 \mapsto 4, h_3 \mapsto 6, l \mapsto 0\} \\ \mu_2 & = & \{h_1 \mapsto 4, h_2 \mapsto 2, h_3 \mapsto 6, l \mapsto 0\} \end{array}$$

For both μ_1 and μ_2, $(h_1 + h_2 + h_3)/3$ evaluates to 4. However:

$$\begin{array}{rcl} \mu_1' & = & \{h_1 \mapsto 2, h_2 \mapsto 4, h_3 \mapsto 6, l \mapsto 2\} \\ \mu_2' & = & \{h_1 \mapsto 4, h_2 \mapsto 2, h_3 \mapsto 6, l \mapsto 4\} \end{array}$$

Example: Electronic Wallet

```
if (declass(h ≥ k, L)) {
  h := h − k;
  l := l + k;
}
```

This program only leaks $h \geq k$ and is accepted by delimited release:

- consider two memories μ_1, μ_2 such that $h \geq k$ evaluates to the same value
- l gets the same value after the execution on μ_1 and μ_2

Example: Electronic Wallet

```
l := 0;
while (n ≥ 0) {
  k := 2^(n−1);
  if (declass(h ≥ k, L)) {
    h := h − k;
    l := l + k;
  }
  n := n − 1
}
```

Let n be the number of bits of h, this program is rejected by delimited release:

- the program is equivalent to $l := h$, which is insecure
- find two memories μ_1, μ_2 which violate delimited release

Typing Delimited Release

We use an extension of a traditional type system for NI.

Two forms of type rules:

- $\Gamma \vdash e : \ell, D$ reading as expression e has label ℓ and declassifies the variables in D under the typing environment Γ
- $\Gamma, pc \vdash c : U, D$ reading as command c is well-typed under the typing environment Γ and the program counter label pc. Moreover, c updates the variables in U and declassifies the variables in D

Intuition: do not update what is eventually declassified!

Typing Rules for Expressions

For expressions, we use rules of the form $\Gamma \vdash e : \ell, D$

$$\Gamma \vdash v : \ell, \emptyset \qquad \Gamma \vdash x : \Gamma(x), \emptyset \qquad \frac{\Gamma \vdash e_1 : \ell, D_1 \qquad \Gamma \vdash e_2 : \ell, D_2}{\Gamma \vdash e_1 \oplus e_2 : \ell, D_1 \cup D_2}$$

$$\frac{\Gamma \vdash e : \ell, D}{\Gamma \vdash \mathbf{declassify}(e, \ell') : \ell', Vars(e)} \qquad \frac{\Gamma \vdash e : \ell, D \qquad \ell \sqsubseteq \ell'}{\Gamma \vdash e : \ell', D}$$

Typing Rules for Commands (1/2)

For commands, we use rules of the form $\Gamma, pc \vdash c : U, D$

$$\Gamma, pc \vdash \textbf{skip} : \emptyset, \emptyset \qquad \frac{\Gamma \vdash e : \ell, D \qquad \ell \sqcup pc \sqsubseteq \Gamma(x)}{\Gamma, pc \vdash x := e : \{x\}, D}$$

$$\frac{\Gamma, pc \vdash c_1 : U_1, D_1 \qquad \Gamma, pc \vdash c_2 : U_2, D_2 \qquad U_1 \cap D_2 = \emptyset}{\Gamma, pc \vdash c_1; c_2 : U_1 \cup U_2, D_1 \cup D_2}$$

Typing Rules for Commands (2/2)

$$\frac{\Gamma \vdash e : \ell, D \qquad \Gamma, \ell \sqcup pc \vdash c_1 : U_1, D_1 \qquad \Gamma, \ell \sqcup pc \vdash c_2 : U_2, D_2}{\Gamma, pc \vdash \textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2 : U_1 \cup U_2, D \cup D_1 \cup D_2}$$

$$\frac{\Gamma \vdash e : \ell, D \qquad \Gamma, \ell \sqcup pc \vdash c : U_1, D_1 \qquad U_1 \cap (D \cup D_1) = \emptyset}{\Gamma, pc \vdash \textbf{while } e \textbf{ do } c : U_1, D \cup D_1}$$

$$\frac{\Gamma, pc \vdash c : U, D \qquad pc' \sqsubseteq pc}{\Gamma, pc' \vdash c : U, D}$$

Integrity and Declassification

When we consider an active attacker, there is a subtle interplay between integrity and declassification.

Intuitively, we desire that the attacker should not be able to influence declassification decisions:

- what is declassified
- whether some information is declassified or not

An active attacker should not be more powerful than a passive attacker!

Examples

Are these programs (intuitively) secure?

Example

$x_{LH} := \textbf{declassify}(y_{HH}, LH)$

Example

$\textbf{if } z_{LH} > 0 \textbf{ then } x_{LH} := \textbf{declassify}(y_{HH}, LH) \textbf{ else skip}$

Example

$\textbf{if } z_{LL} > 0 \textbf{ then } x_{LL} := \textbf{declassify}(y_{HH}, LH) \textbf{ else skip}$

Robust Declassification

Technically, we proceed as follows:

- we extend the syntax of programs with holes, where the attacker can put arbitrary malicious code: $c[\vec{\bullet}]$
- we require the attacker to only mention variables with label LL

Definition

The command $c[\vec{\bullet}]$ has robustness iff for all memories μ_1, μ_2 and all the attacks $\vec{a}_1, \vec{a}_2$ we have that if $\langle c[\vec{a}_1], \mu_1 \rangle \Downarrow \mu_1'$ and $\langle c[\vec{a}_1], \mu_2 \rangle \Downarrow \mu_2'$ with $\mu_1' \approx_{LL} \mu_2'$, then either of the following holds:

1. $c[\vec{a}_2]$ does not terminate on some $\mu \in \{\mu_1, \mu_2\}$
2. $\langle c[\vec{a}_2], \mu_1 \rangle \Downarrow \mu_1''$ and $\langle c[\vec{a}_2], \mu_2 \rangle \Downarrow \mu_2''$ with $\mu_1'' \approx_{LL} \mu_2''$

Examples of Insecure Programs

The following program violates robustness:

$$[\bullet]; \textbf{if } z_{LL} > 0 \textbf{ then } x_{LL} := \textbf{declassify}(y_{HH}, LH) \textbf{ else skip}$$

Pick the following memories and attackers:

$$\begin{array}{lll} \mu_1 & \triangleq & \{x_{LL} \mapsto 0, y_{HH} \mapsto 1, z_{LL} \mapsto 0\} \\ \mu_2 & \triangleq & \{x_{LL} \mapsto 0, y_{HH} \mapsto 2, z_{LL} \mapsto 0\} \end{array} \qquad \begin{array}{lll} a_1 & \triangleq & z_{LL} := -1 \\ a_2 & \triangleq & z_{LL} := +1 \end{array}$$

Examples of Insecure Programs

The following program violates robustness:

$$[\bullet]; z_{LL} := \textbf{declassify}(x_{HL} \geq y_{LL}, LL)$$

Pick the following memories and attackers:

$$\begin{array}{lll} \mu_1 & \triangleq & \{x_{HL} \mapsto 5, y_{LL} \mapsto 4, z_{LL} \mapsto \textbf{true}\} \\ \mu_2 & \triangleq & \{x_{HL} \mapsto 8, y_{LL} \mapsto 4, z_{LL} \mapsto \textbf{true}\} \end{array} \qquad \begin{array}{lll} a_1 & \triangleq & \textbf{skip} \\ a_2 & \triangleq & y_{LL} := 6 \end{array}$$

Enforcing Robustness

Key intuition for robustness:

1. the declassified expression must have integrity
2. declassification must happen on a *pc* with high integrity

$$\frac{\Gamma, LH \vdash e : HH}{\Gamma, LH \vdash \textbf{declassify}(e, LH) : LH} \qquad \Gamma, LH \vdash \bullet$$

Examples Revisited

$$\dfrac{\Gamma, LH \vdash z_{LH} > 0 : LH \qquad \dfrac{\dfrac{\Gamma, LH \vdash y_{HH} : HH}{\Gamma, LH \vdash \textbf{declassify}(y_{HH}, LH) : LH} \qquad LH \sqsubseteq \Gamma(x_{LH})}{\Gamma, LH \vdash x_{LH} := \textbf{declassify}(y_{HH}, LH)}}{\Gamma, LH \vdash \textbf{if } z_{LH} > 0 \textbf{ then } x_{LH} := \textbf{declassify}(y_{HH}, LH) \textbf{ else skip}}$$

$$\dfrac{\Gamma, LH \vdash z_{LL} > 0 : LL \qquad \dfrac{\Gamma, LL \not\vdash \textbf{declassify}(y_{HH}, LH) : LH \qquad LH \sqsubseteq \Gamma(x_{LL})}{\Gamma, LL \vdash x_{LL} := \textbf{declassify}(y_{HH}, LH)}}{\Gamma, LH \vdash \textbf{if } z_{LL} > 0 \textbf{ then } x_{LL} := \textbf{declassify}(y_{HH}, LH) \textbf{ else skip}}$$