Security-Typed Languages
Lecture 4

June 21, 2019

1 Transparent Endorsement

From last time: NI → Robust declassification (NI refines RD). Robust declassification breaks the confidentiality/integrity duality.

To restore duality we define Transparent Endorsement (TE). For an example, see the code fragment in figure 1

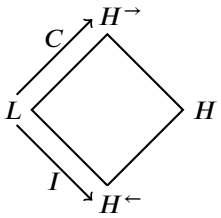


Figure 1: Lattice for TE

A problem can arise when adversary manages to steer password value directly into `check_password` function, abusing downgrading. The problem arises because `pwd` with H label (trusted) can flow into $H^{\rightarrow}$ (untrusted). To solve the problem we can give label L to variable `guess` (untrusted and unconfidential).

This is enforced by the typing rule for *transparent endorsement*:

```
String{H} pwd;
 bool{H←} check_password(String{H→} guess) {
    String{H} endorsed_guess = endorse(guess,H→ to H);
    bool{H} res = (password == endorsed_guess);
    return declassify(res,H to H←);
}
```

Figure 2: Example: password checker

$$\frac{\Gamma(y) \sqsubseteq l_1 \qquad l_2 \sqsubseteq \Gamma(y) \qquad l_1 \sqsubseteq l_2 \sqcup \bigtriangledown(l_1 \sqcup pc)}{pc \vdash x := \text{endorse}(y, l_1 \text{ to } l_2)}$$

$\bigtriangledown$ = voice = maps confidentiality to corresponding integrity

We can describe both robust declassification and transparent endorsement in the same picture:

$s_{11} \quad \sim_L \quad s_{12}$

$\sim_H \qquad\qquad \sim_H$

$s_{21} \quad \sim_L \quad s_{22}$

Definition 1.0.1 (Robust declassification)**.**

$$[\![s_{11}]\!] \approx_L [\![s_{21}]\!] \ \wedge \ \textit{relevant inputs} \Rightarrow [\![s_{12}]\!] \approx_L [\![s_{22}]\!]$$

Definition 1.0.2 (Transparent endorsement)**.**

$$[\![s_{11}]\!] \approx_H [\![s_{12}]\!] \ \wedge \ \textit{relevant inputs} \Rightarrow [\![s_{21}]\!] \approx_H [\![s_{22}]\!]$$

RD + TE ="nonmalleable informationi flow"

Where do $\bigtriangleup$ and $\bigtriangledown$ come from?

FLAM (Arden et al. CSF'15)

1. labels are principals
2. primitive principals (Alice, Bob, p, q, …)
3. prinicpal projections ($p^{\leftarrow}$ integrity projection, $p^{\rightarrow}$ confidentiality)
4. joins and meets on principals $p \wedge q$ (reads as: powers of both p and q), $p \vee q$.

$$\forall p, q. p \wedge q \succeq p \succeq p \vee q$$

where $\succeq$ is a trust ordering. Least powerful principal is $\bot$; most powerful is $\top$. See figure 4

A normal form for principals is $A^{\leftarrow} \wedge B^{\rightarrow}$, where A, B are CNF expressions over primitive principals. Then $\bigtriangleup$ and $\bigtriangledown$ are defined as:

$$\bigtriangleup(A^{\leftarrow} \wedge B^{\rightarrow}) = A^{\rightarrow} \wedge T^{\leftarrow}$$
$$\bigtriangledown(A^{\leftarrow} \wedge B^{\rightarrow}) = B^{\leftarrow}$$
$$\text{Reflection: } \bigtimes (A^{\leftarrow} \wedge B^{\rightarrow}) = B^{\leftarrow} \wedge A^{\rightarrow}$$

If something has label $l \not\sqsubseteq \bigtimes l$, we can't downgrade it nonmalleably. See figure 1.

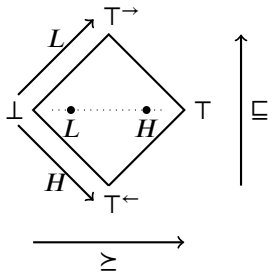


Figure 3: Lattice for FLAM

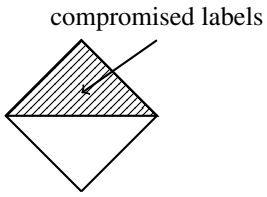


Figure 4: Reflection drawn in lattice

2 Hardware security

There are different layers at the hardware level:

	app code
	libraries
Modern systems:	OS
	ISA
	μ architecture

- Correctness and security depends on having contracts between these different layers
- Classic specifications do not work (Meltdown, spectre)
- Contracts should capture information flow (hyperproperties)
- Contracts should be compositional
- Mandatory vs. discretionary access control

2.1 Example

```
if h1 then
    h2 := l1 //pulls l1 into cache
```

```
else
      h2  := l2
l3  := l1     //false  if  h1  =  true
```

Listing 1: Timing to update l_3 depends on the value of h_1.

2.2 Reference papers for reading

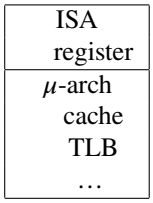
- Zhang/PLDI'12: ISA/M-arch contract that rules out timing channels (in addition to previously discussed leakage)
- Zagieboylo/CSF'19: Detailed ISA contract for realistic ISA supporting nonmalleable downgrading

2.3 IMP: read and write label

Consider imperative language IMP where each command has a *read label* and a *write label*.

Read label and write label properties:

- Read label l_r bounds influences on time taken by instruction
- Write label l_w is a lower bound on effects instruction has on μ-architecture state



It defines two type of properties that processor needs to satisfy. Hardware satisfying these three porperties can reason about information flow for software/hardware composition:

- Architecturlar semantcs (like SOS, $c, \gamma \longrightarrow c', \gamma'$)
- μ-arch semantics: $c, \gamma, E, G \longrightarrow c', \gamma', E', G'$

  where E = the microarchitecture state, G is global (wall-clock) time.

Read-label property:
Execution time should not depend on high state.
Given command $c_{[}l_r, l_w]$
$(\forall x \in vars(c).\gamma_1(x) = \gamma_2(x)) \wedge E_{1\ l_r} E_2) \wedge c_{[}l_r, l_w], \gamma_i, E_i, G \longrightarrow c_i, \gamma_i', E_i', G_i) \Rightarrow G_1 =$

G_2 for $i \in \{1, 2\}$

write-label property:
$l_w \not\sqsubseteq l \wedge c_{[l_r, l_w]}, \gamma, E, G \longrightarrow c', \gamma', E', G' \Rightarrow E \sim_l E'$

Single-step noninterference:
$(\gamma_1 \sim_{-l} \gamma_2 \wedge E_1 \sim_l E_2 \wedge c_{l_r, l_w}, \gamma_i, E_i, G_i \longrightarrow c_i, \gamma_i', E_i', G_i') \Rightarrow E_1' = E_2'$

```
if h1 then
    h2 := l1[L,H] // pulls l1 into cache
else
    h2 := l2[L,H]
l3 := l1[L,L] // false if h1 = true
```

h_1 flows into assignment h_2 of l_1. This is updated code from Example 1.1.

3 HDLs

(Hardware description language)

How do you build *efficient* hardware that *verifiably* satisfies security properties? Use SecVerilog = Verilog + security labels

- Threat model = adversary can see all public memory at every clock cycle.
- Partition cache statically
- Annotations on variables (possibly functions)

Soundness: at each clock tick, no H information leaks to a L variable.
See slides for the rest.