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.

A problem can arise when adversary manages to steer password value directly into check_password function, abusing downgrading. The problem arises because pwd

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

Figure 2: Example: password checker
with H label (trusted) can flow into H\(^{-}\) (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:

\[
\frac{\Gamma(y) \subseteq l_1 \quad l_2 \subseteq \Gamma(y) \quad l_1 \subseteq l_2 \cup \nabla(l_1 \cup pc)}{pc \vdash x := \text{endorse}(y,l_1 \text{ to } l_2)}
\]

\(\nabla = \text{voice} = \text{maps confidentiality to corresponding integrity}\)

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

\[
\begin{align*}
&\, s_{11} \sim L \quad \sim H \\
&\sim H \quad \sim H \\
&\, s_{21} \sim L \quad \sim H
\end{align*}
\]

**Definition 1.0.1** (Robust declassification).

\[
\llbracket s_{11} \rrbracket \approx_L \llbracket s_{21} \rrbracket \land \text{relevant inputs} \Rightarrow \llbracket s_{12} \rrbracket \approx_L \llbracket s_{22} \rrbracket
\]

**Definition 1.0.2** (Transparent endorsement).

\[
\llbracket s_{11} \rrbracket \approx_H \llbracket s_{12} \rrbracket \land \text{relevant inputs} \Rightarrow \llbracket s_{21} \rrbracket \approx_H \llbracket s_{22} \rrbracket
\]

RD + TE = "nonmalleable information i flow"

**Where do \(\triangle\) and \(\nabla\) come from?**

FLAM (Arden et al. CSF’15)

1. labels are principals
2. primitive principals (Alice, Bob, p, q, …)
3. principal projections (\(p^-\) integrity projection, \(p^-\) confidentiality)
4. joins and meets on principals \(p \land q\) (reads as: powers of both \(p\) and \(q\)), \(p \lor q\).

\[
\forall p, q. p \land q \geq p \geq p \lor q
\]

where \(\geq\) is a trust ordering. Least powerful principal is \(\bot\); most powerful is \(\top\).

See figure [1]

A normal form for principals is \(A^- \land B^-\), where \(A, B\) are CNF expressions over primitive principals. Then \(\triangle\) and \(\nabla\) are defined as:

\[
\begin{align*}
\triangle(A^- \land B^-) &= A^- \land T^- \\
\nabla(A^- \land B^-) &= B^-
\end{align*}
\]

Reflection: \(\nabla (A^- \land B^-) = B^- \land A^-\)

If something has label \(l \not\subseteq \nabla l\), we can’t downgrade it nonmalleably. See figure [1]

2
2 Hardware security

There are different layers at the hardware level:

<table>
<thead>
<tr>
<th>Modern systems:</th>
</tr>
</thead>
<tbody>
<tr>
<td>app code</td>
</tr>
<tr>
<td>libraries</td>
</tr>
<tr>
<td>OS</td>
</tr>
<tr>
<td>ISA</td>
</tr>
<tr>
<td>μ architecture</td>
</tr>
</tbody>
</table>

- 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 \( h_1 \) then
\[
    h_2 := l_1 \quad \text{// pulls } l_1 \text{ into cache}
\]
else
\[
    h_2 := l_2
\]
\[
    l_3 := l_1 \quad \text{// false if } h_1 = \text{true}
\]

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)

- Zagiebo/CSF’19: Detailed ISA contract for realistic ISA supporting nonmal-leable 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 \( \mu \)-architecture state

| ISA register |
| \( \mu \)-arch |
| cache |
| TLB |
| … |

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

- Architectural semantics (like SOS, \( c, \gamma \rightarrow c', \gamma' \))
- \( \mu \)-arch semantics: \( c, \gamma, E, G \rightarrow c', \gamma', E', G' \)

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

Read-label property:

Execution time should not depend on high state.
Given command \( c[l_r, l_w] \)
\[
(\forall x \in vars(c). y_1(x) = y_2(x)) \land E_1 \land E_2 \land c[l_r, l_w], y_i, E_i, G_i, G_i' \Rightarrow c_i, y_i', E_i', G_i) \Rightarrow G_1 = G_2 \text{ for } i \in \{1, 2\}
\]

**write-label property:**
\[
l_w \not\subseteq l \land c[l_r, l_w], y, E, G \rightarrow c', y', E', G' \Rightarrow E \rightarrow E'
\]

**Single-step noninterference:**
\[
(y_1 \rightarrow l y_2 \land E_1 \land E_2 \land c[l_r, l_w], y_i, E_i, G_i \rightarrow c_i, y_i', E_i', G_i) \Rightarrow E_1' = E_2'
\]

```haskell
if h_1 then
    h_2 := l_1 [L,H]  // pulls l_1 into cache
else
    h_2 := l_2 [L,H]

l_3 := l_1 [L,L]  // false if h_1 = 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.