Secure Compilation of Constant-Resource Programs

Gilles Barthe    Sandrine Blazy    Rémi Hutin    David Pichardie
Cryptographic Constant-time (CCT)

- A countermeasure to protect against timing side-channels attacks.

- CCT programs must not perform:
  - Secret-dependent branches
  - Secret-dependent memory accesses

- Popular and used by cryptographers:
  - Several cryptographic implementations: AES, Curve25519, RSA, TLS, ...
Observational Non-interference (ONI)

- ONI: generic policy for side-channel leakage. [CSF’18]
  - CCT can be defined as an instance of ONI

- Imperative language with big-step semantics:

  - $\sigma_1 \sim \sigma_2$: both states share the same values for public values and may differ on secret values (indistinguishability).

- A program $p$ is ONI if any pair of executions starting from indistinguishable states $\sigma_1 \sim \sigma_2$ produce the same leakage.

- Intuitively: leakage does not reveal secrets.

Instances of ONI

• CCT is formally defined as an instance of ONI.

• Leakage $\ell$: list of boolean guards and memory accesses.

• Example: semantics rule of if-statement:

$$\langle e, \sigma \rangle \downarrow \text{true}$$

$$\langle \text{if } (e) \{ p_1 \} \{ p_2 \}, \sigma \rangle \downarrow \sigma'$$

$$\langle \text{true} \cdot \ell \rangle$$

• In our work, we consider a different instance of ONI, known as Constant-Resource (CR) or Time-balancing.

• Leakage $\ell$: amount of resources consumed during an execution ($\in \mathbb{N}$).

• Every construct of the language consumes a constant amount of resources. Example rule for sequence:

$$\langle p_1, \sigma \rangle \downarrow \sigma'$$

$$\langle p_2, \sigma' \rangle \downarrow \sigma''$$

$$\langle (p_1; p_2), \sigma \rangle \downarrow \sigma''$$

$$\ell_1 + \ell_2$$
Constant-Resource: a relaxation of CCT

- Has been used to implement cryptographic primitive. Example from s2n, Amazon’s implementation of TLS. [VSTTE’18]

Consider a secret value $x$, bounded: $0 \leq x \leq 32$. Function `update` consumes 1 resource.

- More generally, secret-dependent branch are allowed, as long as branches are balanced.
- $\text{CCT} \subseteq \text{CR}$

### Preservation of ONI during compiler transformation

<table>
<thead>
<tr>
<th>Enforcement / Program repair</th>
<th>Cryptographic Constant-Time</th>
<th>Constant-Resource</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>[ISSTA’18] Meng Wu et al. “Eliminating timing side-channel leaks using program repair”.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>?</td>
<td></td>
</tr>
</tbody>
</table>
### Challenges

#### Compilation

- CR-security relies on fragile **balance** between resources → could easily be broken by common optimizations.

- Our solution: a more flexible security policy $\text{CR}^\#$.

#### Proof methodology

- Existing proof techniques for preservation of other ONI **cannot** be applied.

- The **non-cancelation** property does not hold for resource leakage ($\mathbb{N}$).

$$\ell_1 + \ell'_1 = \ell_2 + \ell'_2 \Rightarrow \ell_1 = \ell_2 \land \ell'_1 = \ell'_2$$

- Intuitively:

$$CR(p_1; p_2) \not\Rightarrow CR(p_1) \land CR(p_2)$$
1. Example

2. Motivate and introduce CR#

3. Present our methodology
Example: Common Subexpression Elimination (CSE)

Resource model:
addition costs 1
multiplication costs 2

```plaintext
if (cond) {
    x = a*b;
    y = (a*b)+c+d;
} else {
    x = a+b;
    y = (a+b)*c*d;
}
```

CSE

```plaintext
if (cond) {
    x = a*b;
    y = x+c+d;
} else {
    x = a+b;
    y = x*c*d;
}
```

CSE#

δ: padding operator

```plaintext
δ(n), σ \downarrow σ
```

δ(2):

```plaintext
δ(2);
        x = a*b;
        y = x+c+d;
} else {
    δ(1);
    x = a+b;
    y = x*c*d;
}
```

min

```plaintext
if (cond) {
    δ(2-1);
    x = a*b;
    y = x+c+d;
} else {
    δ(1-1);
    x = a+b;
    y = x*c*d;
}
```

Not balanced anymore

2 additions and 2 multiplications in both branches → balanced

Still balanced, thanks to padding

δ: padding operator

\( \delta(n), \sigma \downarrow \sigma \)

Resource model:
addition costs 1
multiplication costs 2

```plaintext
if (cond) {
    x = a*b;
    y = x+c+d;
} else {
    x = a+b;
    y = x*c*d;
}
```
A secret-aware compiler

• Our approach introduces padding and restricts the compiler
  → only necessary in secret-dependent branches.

• First approach: security type-system.
  • Pros: keeps precise track of security levels.
  • Cons: does not scale to realistic compiler.

• Our approach: syntactic annotation, called atomic.
  • Inspired from parallel computing (barriers).
  • Easily introduced by a previous analysis at source level.
  • Statically identify high security parts of the program.
  • Compiler only restricted in annotated parts.
## Atomic annotations

### Compiler

- Restricted (by introducing padding) **inside** atomic annotations.
- Unrestricted elsewhere.

### Flexible policy

- **New policy: CR#**
- Expects CR behavior inside atomic annotations.
- Elsewhere, secret-dependent branches are not allowed (CCT-like behavior).

```java
if (public) {
    ...
} else {
    ...
}

if (secret) {
    ...
} else {
    ...
}
```

[Diagram of atomic annotations and code example]
Formal definition of CR#

- CR# is defined as an instance of ONI.

- Leakage $\ell$:
  \[
  \ell = (f, q)
  \]
  - $f$: control-flow, list of boolean guard, CCT-like leakage
  - $q$: resources, CR-like leakage

- CR#-security expects control-flow and resource consumption to be independent from secrets.

- Relaxed by atomic semantics:
  \[
  \begin{align*}
  &\langle p, \sigma \rangle \xrightarrow{(f,q)} \sigma' \\
  \langle p, \sigma \rangle \xrightarrow{(\epsilon,q)} \sigma' \\
  \end{align*}
  \]

CR# is a flexible mix between CCT and CR

CCT $\subseteq$ CR# $\subseteq$ CR
Methodology

- We decompose a control-flow preserving (CSE, constant prop., ...) transformation $T$ as $\text{min} \circ T^\#$:

\[
\begin{align*}
\text{if (public)} & \{ \\
& \quad \cdots \\
& \} \quad \text{else} \{ \\
& \quad \cdots \\
& \}
\end{align*}
\]

\[
\begin{align*}
\text{if (secret)} & \{ \\
& \quad \cdots \\
& \} \quad \text{else} \{ \\
& \quad \cdots \\
& \}
\end{align*}
\]

Proved CR$^\#$-preserving as it preserves leakage.

\[
\begin{align*}
\text{if (public)} & \{ \\
& \quad \delta(T_1); \cdots \\
& \} \quad \text{else} \{ \\
& \quad \delta(T_2); \cdots \\
& \}
\end{align*}
\]

\[
\begin{align*}
\text{if (secret)} & \{ \\
& \quad \delta(T_3); \cdots \\
& \} \quad \text{else} \{ \\
& \quad \delta(T_4); \cdots \\
& \}
\end{align*}
\]

Proved CR$^\#$-preserving (main proof effort).

\[
\begin{align*}
\text{if (public)} & \{ \\
& \quad \cdots \\
& \}
\end{align*}
\]

\[
\begin{align*}
\text{if (secret)} & \{ \\
& \quad \delta(T_3-T); \cdots \\
& \} \quad \text{else} \{ \\
& \quad \delta(T_4-T); \cdots \\
& \}
\end{align*}
\]
• We presented a security policy called CR#, a flexible mix between CCT and CR, that relies on atomic annotations.

• We developed a proof methodology to prove that a transformation preserves CR#, and applied it to generic control-flow preserving transformations.

• All our results are mechanically verified with the Coq proof assistant.