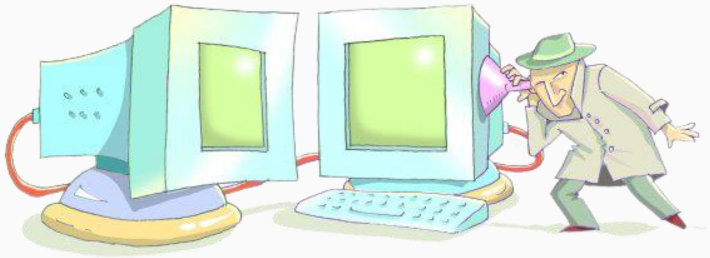
Secure Compilation of Constant-Resource Programs

Gilles Barthe Sandrine Blazy **Rémi Hutin** David Pichardie

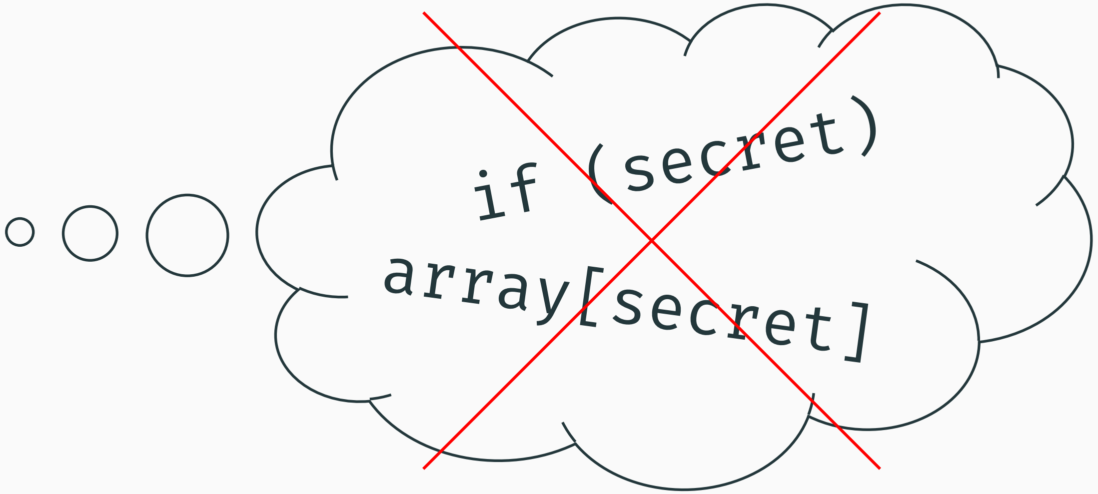
erc UMR IRISA Inria INVENTEURS DU MONDE NUMÉRIQUE UNIVERSITÉ DE RENNES 1 ueb MAX-PLANCK-INSTITUT FÜR SICHERHEIT UND PRIVATSPHÄRE institute IMdea ENS rennes

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



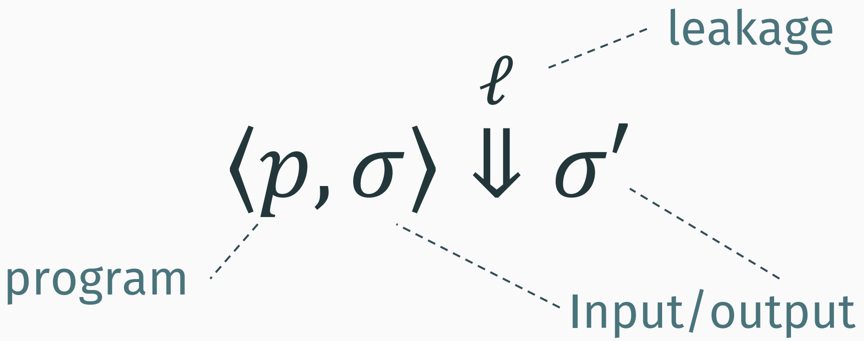
~~`if (secret)`~~

~~`array[secret]`~~

- 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



$$\langle p, \sigma \rangle \overset{\ell}{\Downarrow} \sigma'$$

- 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.

$$\mathrm{ONI}(p):$$

$$\begin{array}{l} \langle p, \sigma_1 \rangle \overset{\ell_1}{\Downarrow} \sigma_1' \\ \wr \\ \langle p, \sigma_2 \rangle \overset{\ell_2}{\Downarrow} \sigma_2' \end{array} \quad \text{implies } \ell_1 = \ell_2$$

[CSF'18] Gilles Barthe et al. "Secure Compilation of Side-channel Countermeasures: the case of Cryptographic Constant-Time"

Instances of ONI

- CCT is formally defined as an instance of ONI.
- Leakage ℓ: list of boolean guards and memory accesses.
- Example: semantics rule of if-statement:

$$\frac{\langle e, \sigma \rangle \Downarrow true \qquad \langle p_1, \sigma \rangle \overset{\ell}{\Downarrow} \sigma'}{\langle \boldsymbol{if}\ (e)\ \{\, p_1\}\ \{p_2\}, \sigma \rangle \overset{true \cdot \ell}{\Downarrow} \sigma'}$$

- In our work, we consider a different instance of ONI, known as Constant-Resource (CR) or Time-balancing.
- Leakage ℓ: amount of resources consumed during an execution ($\in \mathbb{N}$).
- Every construct of the language consumes a constant amout of resources. Example rule for sequence:

$$\frac{\langle p_1, \sigma \rangle \overset{\ell_1}{\Downarrow} \sigma' \qquad \langle p_2, \sigma' \rangle \overset{\ell_2}{\Downarrow} \sigma''}{\langle (p_1; p_2), \sigma \rangle \overset{\ell_1 + \ell_2}{\Downarrow} \sigma''}$$

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.

```
repeat (x) { update(); }
```

Consumes x resources

Repair →

```
repeat (x) { update(); }
repeat (32-x) { dummyUpdate(); }
```

Consumes 32 resources

- More generally, secret-dependent branch are allowed, as long as branches are balanced.
- CCT $\sqsubseteq$ CR

[VSTTE'18] Athanasiou, Konstantinos, et al. "Sidetrail: Verifying time-balancing of cryptosystems."

Preservation of ONI during compiler transformation

	Cryptographic Constant-Time	**Constant-Resource**
Enforcement / Program repair	[PLDI’19] Sunjay Cauligi et al. “FaCT: a DSL for timing-sensitive computation”.	[POPL’00] Johan Agat. “Transforming out timing leaks”. [ISSTA’18] Meng Wu et al. “Eliminating timing side-channel leaks using program repair”. [*S&P. 2017*] Mario Dehesa-Azuara et al. “Verifying and synthesizing constant-resource implementations with types”.
Preservation	[CCS’17] José Bacelar Almeida et al. “Jasmin: High-assurance and high-speed cryptography”. [POPL’20] Gilles Barthe et al. “Formal verification of a constant-time preserving C compiler”.	**?**

Challenges

Compilation

- CR-security relies on fragile balance between resources → could easily be broken by common optimizations.
- Our solution: a more flexible security policy CR$^{\#}$.

Proof methodology

- Existing proof techniques for preservation of other ONI cannot be applied.
- The non-cancelation property does not hold for resource leakage ($\mathbb{N}$).

$$\ell_1 + \ell_1' = \ell_2 + \ell_2' \implies \ell_1 = \ell_2 \land \ell_1' = \ell_2'$$

- 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

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

2 additions and 2 multiplications in both branches → balanced

$\xrightarrow{\text{CSE}}$

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

Not balanced anymore

$\xrightarrow{\text{CSE}^{\#}}$

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

Still balanced, thanks to padding

δ : padding operator

$$\langle \boldsymbol{\delta}(n), \sigma \rangle \overset{n}{\Downarrow} \sigma$$

$\xrightarrow{\text{min}}$

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

```
if (public) {
        …
} else {
        …
}
if (secret) {
        …
} else {
        …
}
if (public) {
        …
} else {
        …
}
```

Atomic annotation

Flexible policy

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

Formal definition of CR$^{\#}$

- CR$^{\#}$ is defined as an instance of ONI.
- Leakage ℓ:

$$\ell = (f, q)$$

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:

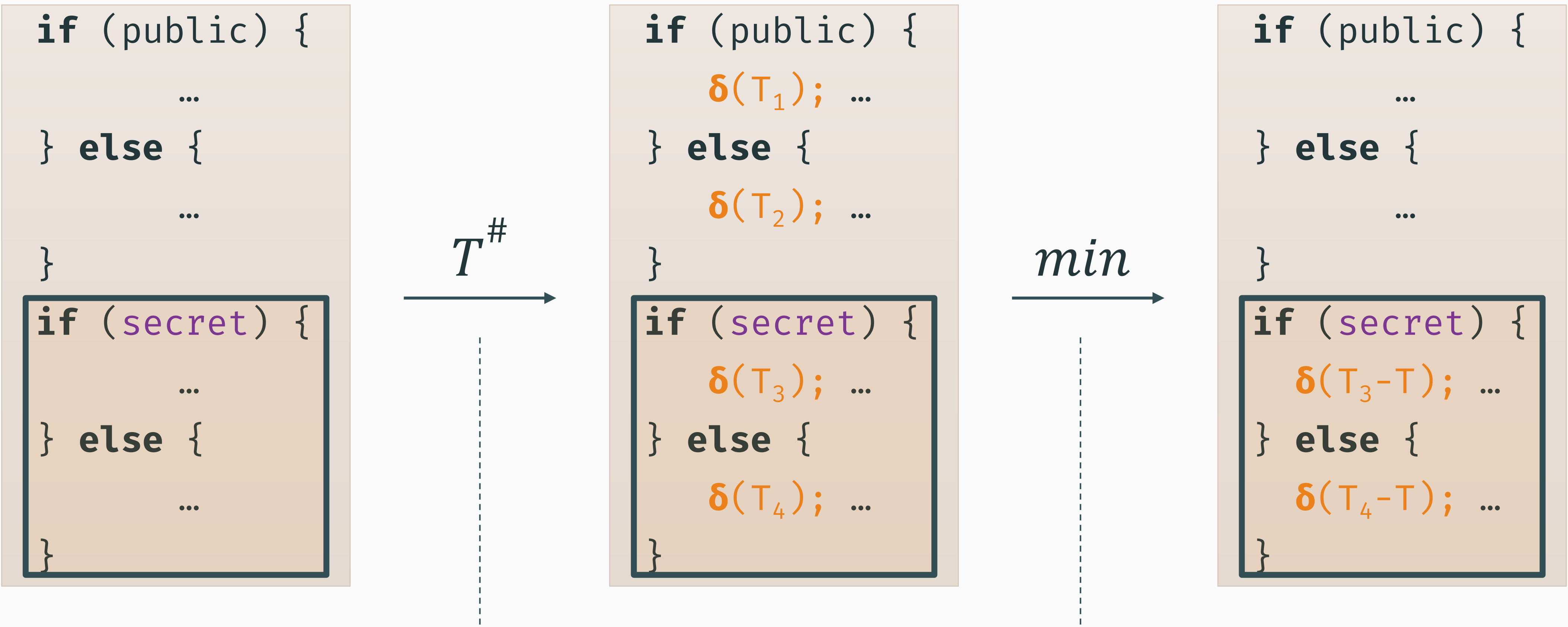
$$\frac{\langle p, \sigma \rangle \overset{(f, q)}{\Downarrow} \sigma'}{\langle \boxed{p}, \sigma \rangle \overset{(\epsilon, q)}{\Downarrow} \sigma'}$$

CR$^{\#}$ is a flexible mix between CCT and CR

$$\text{CCT} \subseteq \text{CR}^{\#} \subseteq \text{CR}$$

Methodology

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



```
if (public) {
      …
} else {
      …
}
if (secret) {
      …
} else {
      …
}
```

$\xrightarrow{T^{\#}}$

```
if (public) {
   δ(T1); …
} else {
   δ(T2); …
}
if (secret) {
   δ(T3); …
} else {
   δ(T4); …
}
```

$\xrightarrow{min}$

```
if (public) {
      …
} else {
      …
}
if (secret) {
  δ(T3-T); …
} else {
  δ(T4-T); …
}
```

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

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

Conclusion

- 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.