How to Compute under AC^0 Leakage without Secure Hardware

Guy Rothblum

Microsoft Research
Silicon Valley

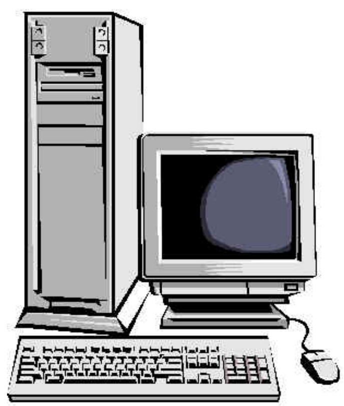
Protecting Sensitive Computations from Leakage/Side-Channel Attacks

Sensitive computations:

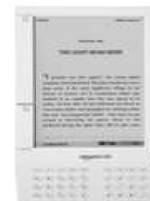
- Cryptographic Algorithms
  - Secret Key

- Proprietary Search Algorithm, Private Medical Data Base Processing…
  - Secret Program, Data

… are Performed Remotely



Mobile Devices



Remote Computing

Google Docs

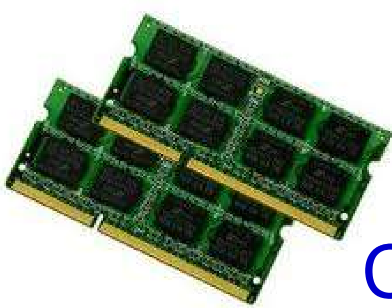
amazon web services

Windows Azure Platform

Computation Internals Might Leak

Power
Consumption
[Kocher et al. 98]

EM Radiation
[Quisquater 01]



Cache [Kocher 96]

Timing [Kocher 96]

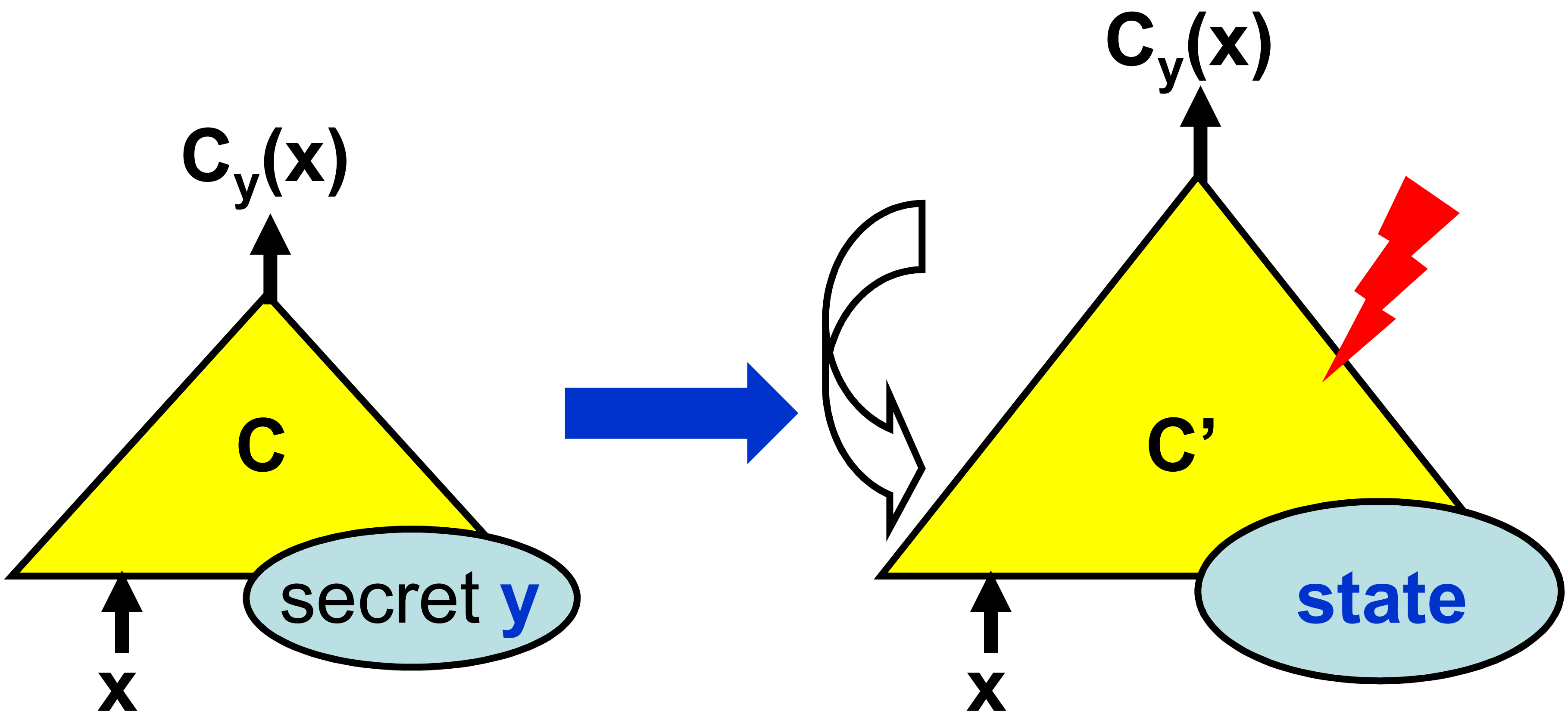
Two Approaches to Fighting Leakage Attacks

- Consider leakage at **design time** [AGV09,…]
  build systems secure against leakage attacks

HOLY GRAIL

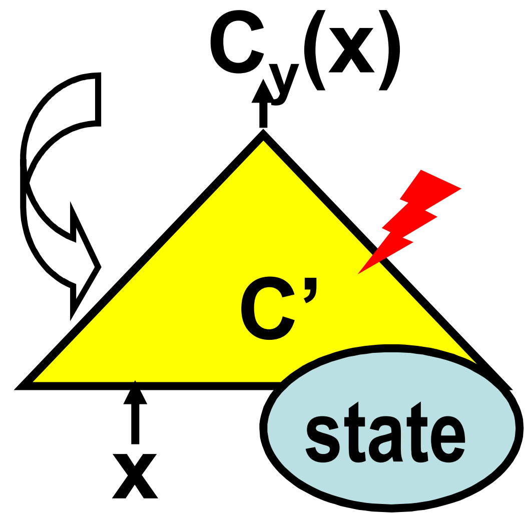
- **“Leakage resilience compiler”** [GO96, ISW03,…]
  transform **any** algorithm so that,
  even under leakage,
  no more than black-box behavior is exposed

Our Goal: Leakage-Resilience Compiler



Even given leakage,
execution "looks like"
black-box access to $C_y(x)$

Offline/Online Leakage Model



Offline (only once): no leakage

Process C and y

$s_1 \leftarrow \mathrm{Init}(C,y,r_0)$

Online, in each execution $t \leftarrow 1,2,3\ldots$

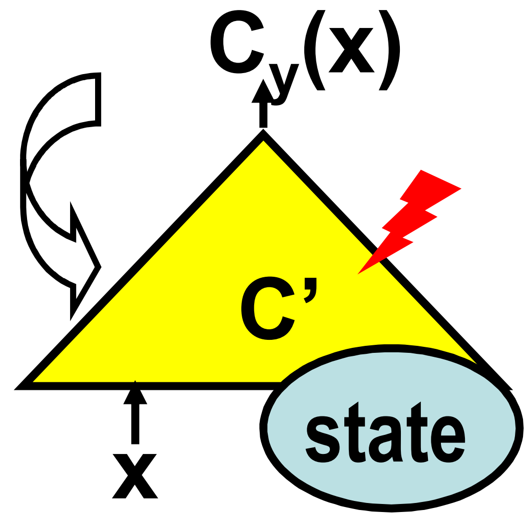
Adv chooses input x_t

$\mathrm{output}_t \leftarrow C'(x_t,s_t,r_t)$, $s_{t+1} \leftarrow \mathrm{Update}(s_t,r_t)$

Adv observes: output_t + $\mathrm{Leakage}_t(x_t,s_t,r_t)$

$\mathrm{Leakage}_t$: leakage function chosen from class of permissible functions

Offline/Online Leakage Model



Offline (only once): no leakage

Process **C** and **y**

$\mathbf{s_1 \leftarrow Init(C,y,r_0)}$

Online, in each execution $\mathbf{t \leftarrow 1,2,3\ldots}$

Adv chooses input $\mathbf{x_t}$

$\mathbf{output_t \leftarrow C'(x_t,s_t,r_t),\ s_{t+1} \leftarrow Update(s_t,r_t)}$

Adv observes: $\mathbf{output_t}$ + $\mathbf{Leakage_t(x_t,s_t,r_t)}$

$\mathbf{Leakage_t}$: **In this work** - **$\mathbf{AC^0}$** function with bounded output length

What is AC^0?

A function **L** is in $\mathbf{AC^0}$ if it can be computed by a **poly-size O(1) depth boolean circuit** with unbounded fan-in AND, OR (and NOT) gates

Some known lower bounds on $\mathbf{AC^0}$

- can't compute parity of **n** bits [H86]
- can't compute inner product of **n**-bit vectors
- can't **"compress"** parity or inner product [HN10,DI06]

New Result: Compiler for AC^0 Leakage

Can transform any poly time C_y into C'

On security parameter κ:

1. Leakage_t is AC^0, output bound = $\lambda(\kappa)$ bits
2. $|C'|=O(\kappa^3 \cdot |C|)$
3. Assuming the **λ-IPPP** assumption, exists simulator **SIM**, s.t.

$$\mathbf{VIEW}_{\text{Leakage}}(C') \approx \mathbf{SIM}^{C_y}$$

λ-IPPP Assumption

Known limits on power of **AC^0** circuits: [H86,DI06]
given **$x,y \in \{0,1\}^\kappa$**, can't compute or compress **$\langle x,y \rangle$** using an **AC^0** circuit

λ-**I**nner **P**roduct w. **P**re-**P**rocessing (**IPPP**) assump

1. poly time to pre-process **$x \Rightarrow f(x)$**
2. poly time to pre-process **$y \Rightarrow g(y)$**
3. given **$f(x)$**,**$g(y)$**, can't compute or compress **$\langle x,y \rangle$** to **$\lambda(n)$** bits using an **AC^0** circuit

Long standing open problem in complexity theory

New Result: Compiler for AC^0 Leakage

Can transform any poly time **C_y** into **C'**

On security parameter **κ**:

1. **Leakage_t** is **AC^0**, output bound = **$\lambda(\kappa)$** bits
2. **$|C'|=O(\kappa^3 \cdot |C|)$**
3. Assuming the **λ-IPPP** assumption, exists simulator **SIM**, s.t.

$$\mathbf{VIEW}_{\mathbf{Leakage}}(\mathbf{C'}) \approx \mathbf{SIM}^{\mathbf{C_y}}$$

Prior Work on General Compilers

"Wire-probe" (either/or) leakage functions

[ISW 03],[A10]	no hardware, unconditional

"Local" (OC) leakage functions [MR04]

[GR10],[JV10]	secure hardware + crypto
[DF12]	secure hardware, unconditional
[GR12]	no hardware, unconditional

AC^0 leakage functions

[FRRTV10]	secure hardware, unconditional

Compiler: High-Level View
(a la [ISW03],[FRRTV10])

- **Init** – "encrypt" bits of **y**
  Enc(b) ⇒ "bundle of bits" - random vector, parity **b**
    (**AC^0** leakage cannot determine parity)

- **Single execution**
  Homomorphically compute on "bundles"
    (computation not in **AC^0**, but resists **AC^0** leakage,
    secure hardware used for "blinding")

- **Multiple executions**
  leakage on bundles encrypting **y** might accumulate
    (secure hardware used to "refresh" bundles)

[FRRTV10] Secure Hardware

Functionality:

generates a random bundle with parity **0**
assume: no leakage on generation procedure

Security:

simulator can create view where the bundle parity is **1**, **AC^0** leakage can't tell the difference

Uses in the construction:

- "blinding" homomorphic computations
- refreshing **y** bundles between executions

New Tool: "Bundle Bank"
(a la [GR12])

"Realize secure hardware", even though
leakage operates also on generation procedure

Functionality:
generate bundles $\mathbf{v_1,v_2,\ldots,v_T}$, s.t. parity $\mathbf{v_i=0}$

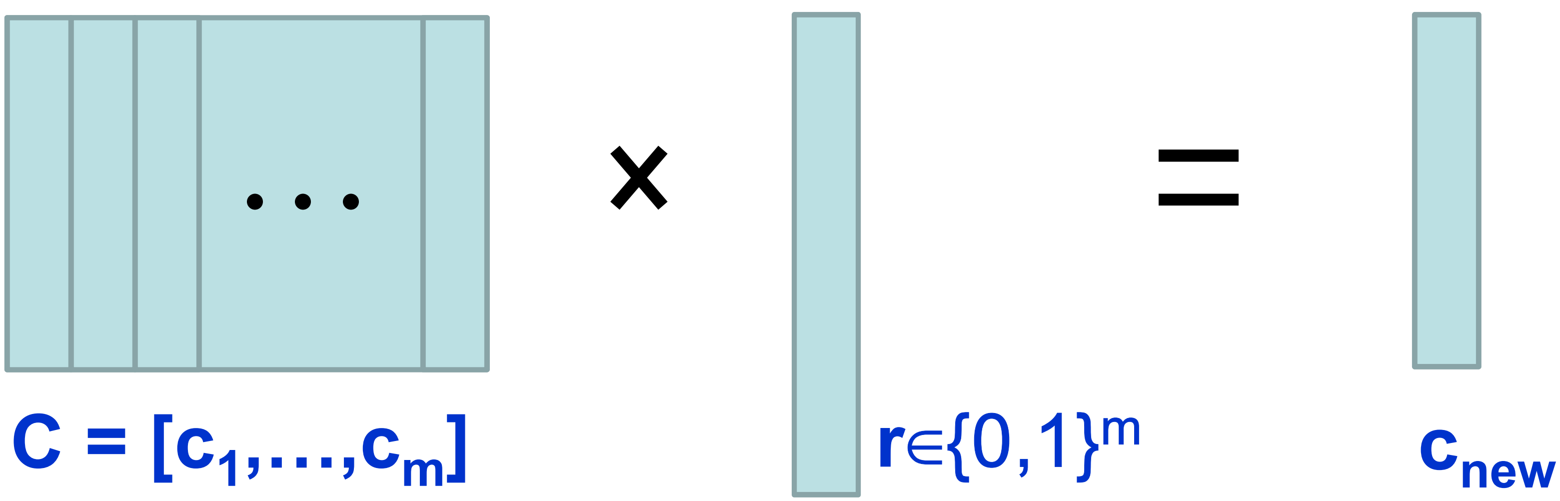
Security:
Simulator on input ($\mathbf{b_1}$, $\mathbf{b_2}$,…,$\mathbf{b_T}$)
generate bundles $\mathbf{v_1,v_2,\ldots,v_T}$, s.t. parity $\mathbf{v_i}=\mathbf{b_i}$

$\mathbf{AC^0}$ leakage on **REAL** and **SIM** is statistically close

Generating One New Bundle

Init (no leakage):
choose $\mathbf{m}$ bundles $\mathbf{c_1 \ldots c_m}$ with parity $\mathbf{0}$

Generating $\mathbf{c_{new}}$ (under leakage):
take random linear combination $\mathbf{r}$



Simulated Generation

Init (no leakage):

~~choose m bundles $c_1 \ldots c_m$ with parity 0~~

parities are **random**: $x \in \{0,1\}^m$

Generating c_{new} (under leakage):

~~take random linear combination r~~

take **biased** linear combination r s.t. $\langle x,r \rangle = b$
($\Rightarrow$ c_{new} parity equals b)

Secure?

AC^0 leakage can't tell if c_i's have parity 0 or 1, and can't tell if r used in generation is biased

Bundle Bank Security

Consider **AC^0** leakage on **REAL** and **SIM** generating a sequence of **0**-bundles

Want: **AC^0** security reduction from parity to distinguishing **REAL** and **SIM**

Obstacle: generation procedure not in **AC^0** (nor are many other computations in construction)

Our main technical contribution:

AC^0 security reduction from IPPP to distinguishing leakage on **REAL** and **SIM**

Why IPPP? Use pre-processing to set up views

THANK YOU!

- Compiler transforms **any** computation into one that resists **AC^0** leakage (under IPPP assumption)
- Strong black-box security
- Secure hardware is not needed

Questions

- IPPP assumption
- Constant leakage rate
- Connections to obfuscation
- Other leakage classes