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

Windows Azure Platform
Computation Internals Might Leak

EM Radiation [Quisquater 01]

Cache [Kocher 96]

Timing [Kocher 96]

Power Consumption [Kocher et al. 98]
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 (only once): no leakage

Process $C$ and $y$

\[ s_1 \leftarrow \text{Init}(C,y,r_0) \]

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

Adv chooses input $x_t$

\[ \text{output}_t \leftarrow C'(x_t,s_t,r_t), \ s_{t+1} \leftarrow \text{Update}(s_t,r_t) \]

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

$\text{Leakage}_t$: leakage function chosen from class of permissible functions
Offline/Online Leakage Model

**Offline** (only once): no leakage

Process $C$ and $y$

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

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

Adv chooses input $x_t$

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

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

**Leakage$_t$**: In this work - $AC^0$ function with bounded output length
What is $\text{AC}^0$?

A function $L$ is in $\text{AC}^0$ 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 $\text{AC}^0$

- 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 AC⁰ Leakage

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

On security parameter $\kappa$:

1. Leakageₜ is AC⁰, output bound = $\lambda(\kappa)$ bits
2. $|C'|=O(\kappa^3 \cdot |C|)$
3. Assuming the $\lambda$-IPPP assumption, exists simulator $SIM$, s.t.
   
   $VIEW_{Leakage}(C') \approx SIM^{C_y}$
**λ-IPPP Assumption**

Known limits on power of \( \text{AC}^0 \) circuits: [H86,DI06]

- given \( x,y \in \{0,1\}^k \), can’t compute or compress \( <x,y> \) using an \( \text{AC}^0 \) circuit

**λ-Inner Product w. Pre-Processing (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 \( <x,y> \) to \( \lambda(n) \) bits using an \( \text{AC}^0 \) circuit

Long standing open problem in complexity theory
New Result: Compiler for $\text{AC}^0$ Leakage

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

On security parameter $\kappa$:

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

$$\text{VIEW}_{\text{Leakage}}(C') \approx SIM^{C_y}$$
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 $y$
  $\text{Enc}(b) \Rightarrow \text{“bundle of bits” - random vector, parity } b$
  ($\text{AC}^0$ leakage cannot determine parity)

- **Single execution**
  Homomorphically compute on “bundles”
  (computation not in $\text{AC}^0$, but resists $\text{AC}^0$ 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 $v_1, v_2, \ldots, v_T$, s.t. parity $v_i = 0$

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

$AC^0$ leakage on REAL and SIM is statistically close
Generating One New Bundle

**Init** (no leakage):
choose \( m \) bundles \( c_1 \ldots c_m \) with parity \( 0 \)

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

\[
C = \begin{bmatrix} c_1, \ldots, c_m \end{bmatrix}
\]

\[
\begin{array}{c}
\mathbf{\ldots} \\
\hline
\end{array}
\times
\begin{array}{c}
\mathbf{r} \\
\in \{0,1\}^m
\end{array}
= \begin{array}{c}
c_{\text{new}} \\
\end{array}
\]
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 $c_{\text{new}}$** (under leakage):
- take random linear combination $r$

- take biased linear combination $r$ s.t. $\langle x, r \rangle = b$
  ($\Rightarrow c_{\text{new}}$ 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 $\text{AC}^0$ leakage on \textsc{REAL} and \textsc{SIM} generating a sequence of 0-bundles

\textbf{Want:} $\text{AC}^0$ security reduction from parity to distinguishing \textsc{REAL} and \textsc{SIM}

\textbf{Obstacle:} generation procedure not in $\text{AC}^0$ (nor are many other computations in construction)

\textbf{Our main technical contribution:}

$\text{AC}^0$ security reduction from IPPP to distinguishing leakage on \textsc{REAL} and \textsc{SIM}

Why IPPP? Use pre-processing to set up views
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