Génération de code au runtime pour la sécurité des systèmes embarqués

COGTO

RESSI – Toulouse
10/05/2016

Damien Couroussé
CEA – LIST / LIALP – Grenoble Université Alpes
CEA LIST-DACLE
Abderahmanne Seriai, Damien Couroussé, Hassan Noura, Nicolas Belleville, Thierno Barry
- Bringing the deGoal framework
- Compilation & runtime code generation

INRIA Rennes, Tamis team
Hélène Le Bouder, Jean-Louis Lanet
- JavaCards
- Physical & logical attacks, software countermeasures

ENSMSE / CEA Tech DPACA - SAS laboratory
Bruno Robisson, Olivier Potin, Karim Abdelatif, Philippe Jaillon
- Physical attacks, HW/SW countermeasures
- Experimental validation

CYBER-PHYSICAL ATTACKS

C1–VCC  C5–GND
C2–RST  C6–VPP
C3–CLK  C7–I/O
C4–      C8–

cyber

Courtesy of Sylvain Guilley, Télécom ParisTech - Secure-IC
powerline analysis  underfeeding

tearing  EMA  power glitch

overclocking  cyber

clock glitch  EMI

heating  laser

Courtesy of Sylvain Guilley, Télécom ParisTech - Secure-IC
BESTIARY OF EMBEDDED SYSTEMS

... IN NEED FOR SECURITY CAPABILITIES

Smart Card

Secure Element inside...

... And many other things
Target inspection
- HW inspection: decapsulation, abrasion, chemical etching, memory extraction, etc.
- SW inspection: debug, memory dumps, code analysis, etc.

Intrusive / active attacks: fault injection
- under/over voltage drops
- iom / laser beam, optical illumination
- glitch attacks
  ...

Observation attacks: side channel attacks
- Electromagnetic analysis
- Power analysis
- Timing attacks
- Acoustic analysis
  ...

APPROXIMATE TYPOLOGY OF PHYSICAL ATTACKS
SIMPLE POWER ANALYSIS (SPA)

SPA on AES [Kocher, 2011]

The AES rounds are « clearly » visible
SPA on RSA [Kocher, 2011]

Direct access to key contents:
- bit 0 = square
- bit 1 = square, multiply
select $n$ clear inputs $\Rightarrow$ record $n$ observations from the target
compute $n$ intermediate values, for each possible key values
compute \{power/EM/timing...\} estimation from the intermediate values
compute the correlation with the observation traces, for each observation sample

---

**Figure 6.3.** All rows of $R$. Key hypothesis 225 is plotted in black, while all other key hypotheses are plotted in gray.

**Figure 6.4.** The column of $R$ at 13.8 µs for different numbers of traces. Key hypotheses 225 is plotted in black.
APPROXIMATE TYPOLOGY OF PHYSICAL ATTACKS

- **Target inspection**
  - HW inspection: decapsulation, abrasion, chemical etching, memory extraction, etc.
  - SW inspection: debug, memory dumps, code analysis, etc.

- **Fault injection attacks**
  - under/over voltage drops
  - ion / laser beam, EM perturbation
  - optical illumination
  - glitch attacks
  - ...

- **Side channel attacks**
  - Electromagnetic analysis
  - Power analysis
  - Acoustic analysis
  - Timing attacks

=> **spatial and temporal sensibility**
An attack is usually split between:

1. A first step attack:
   - global inspection of the target
   - identification of the security components involved (HW/SW)
   - identification of weaknesses

2. A second step attack:
   - focused attack
   - on an identified potential weakness
Definition

Regularly changing the behavior of a (secured) component, at runtime, while maintaining unchanged its functional properties, with runtime code generation.

Diagram:
- Secured Component
  - inputs
  - alea
  - outputs
- Runtime Code Generator
  - inputs
  - alea
- Polymorphic Instance
  - outputs
POLYMORPHIC RUNTIME CODE GENERATION

Definition

- Regularly changing the behavior of a (secured) component, at runtime, while maintaining unchanged its functional properties, with runtime code generation

- Protection against reverse engineering of SW
  - the secured code is not available before runtime
  - the secured code regularly changes its form (code generation interval $\omega$)

- Protection against physical attacks
  - polymorphism changes the spatial and temporal properties of the secured code: side channel & fault attacks
  - Compatible with State-of-the-Art HW & SW Countermeasures

- deGoal: runtime code generation for embedded systems
  - fast code generation
  - tiny memory footprint: proof of concept on TI's MSP430 (512 B RAM)
  - Easy targeting of application-specific instructions or HW features
COUNTERMEASURES & POLYMORPHISM

State of the Art

- Random register renaming [May 2011a, Agosta 2012]
- Out-of-Order execution
  - At the instruction level [May 2011b, Bayrak 2012]
  - At the control flow level [Agosta 2014, Crane 2015]
- A few proof-of-concept implementations, not suitable for embedded devices [Amarilli 2011, Amarilli 2011, Agosta 2012]

Our approach

- Pure software → portability, genericity
- Combination of *all* the polymorphic levers found in the state of the art,
  - Currently at the basic block level
- Modest overhead (execution time & memory footprint)
- With runtime code generation
APPLICATION TO AES

Reference version:

AES 8 bits.c → arm-none-eabi-gcc → Binary image

Polymorphic version:

Polymorphic code generation library

AES.cdg → deGoal + arm-none-eabi-gcc → Binary image

Polymorphic code generator

Runtime code generation

Polymorphic instance of AES
IMPACT OF POLYMORPHISM ON CPA

Reference version

![Graph showing correlation coefficient over time and number of traces.](image)
IMPACT OF POLYMORPHISM ON CPA

Adding a bit of temporal dispersion
IMPACT OF POLYMORPHISM ON CPA

Effect of the code generation interval

Reference implementation

Polymorphic version, code generation intervall: 500

Distinguish threshold = 39 traces
Key byte 10

Distinguish threshold = 89 traces
Key byte 02
IMPACT OF POLYMORPHISM ON CPA

Polymorphic version
code generation interval: 20

Distinguish threshold > 10000 traces
Key byte 02

Polymorphic version, code generation interval: 500

Distinguish threshold = 89 traces
Key byte 02
Polymorphic code generation library

AES.cdg

deGoal + arm-none-eabi-gcc

-Wl,--gc-sections

Binary image

Runtime code generation

Polymorphic instance of AES

Overhead due to runtime code generation

Overhead due to the generated code
### AES 8-BIT. PERFORMANCE OVERHEAD

\[ k = \frac{t_{\text{gen}} + \omega \times t_{\text{poly}}}{\omega \times t_{\text{ref}}} \]

- **k**: performance overhead factor
- **\( \omega \)**: runtime code generation interval

<table>
<thead>
<tr>
<th></th>
<th>AddRoundKey</th>
<th>SubBytes</th>
<th>All round functions</th>
<th>Agosta et al. (2012)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>k Min.</td>
<td>k Avg.</td>
<td>k Max.</td>
<td>k Min.</td>
</tr>
<tr>
<td>( \omega = 1 )</td>
<td>3.16</td>
<td>4.91</td>
<td>6.37</td>
<td>5.81</td>
</tr>
<tr>
<td>( \omega = 10 )</td>
<td>1.32</td>
<td>1.50</td>
<td>1.66</td>
<td>1.59</td>
</tr>
<tr>
<td>( \omega = 100 )</td>
<td>1.09</td>
<td>1.16</td>
<td>1.22</td>
<td>1.16</td>
</tr>
<tr>
<td>( \omega = 1000 )</td>
<td>1.09</td>
<td>1.13</td>
<td>1.18</td>
<td>1.16</td>
</tr>
<tr>
<td>( \omega = 10000 )</td>
<td>1.05</td>
<td>1.12</td>
<td>1.18</td>
<td>1.11</td>
</tr>
</tbody>
</table>

- **Variable performance results** according to
  - Settings of the polymorphic code generator
  - model of noise insertion
- Code is slower when executed in RAM (memory accesses)
- Room for performance improvements
  - The non-polymorphic generated code is slower than the reference

*Extrapolated values*
# AES 8 BIT. MEMORY FOOTPRINT

<table>
<thead>
<tr>
<th></th>
<th>text</th>
<th>data</th>
<th>bss</th>
<th>total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unprotected</td>
<td>4857</td>
<td>52</td>
<td>1168</td>
<td>6077</td>
</tr>
<tr>
<td>AddRoundKey only</td>
<td>8785</td>
<td>56</td>
<td>2980</td>
<td>11821</td>
</tr>
<tr>
<td>SubBytes only</td>
<td>7833</td>
<td>56</td>
<td>2980</td>
<td>10869</td>
</tr>
<tr>
<td>Full polymorphic</td>
<td>14913</td>
<td>56</td>
<td>6052</td>
<td>21021</td>
</tr>
</tbody>
</table>
WHAT’S NEXT?

Experimental evaluation

- State-of-the-art side channel attacks
  - Synchronisation
  - Filtering
- Faults
  - Topic to be opened
- Vulnerability of the code generator?

Open questions

- Certification of polymorphic code? Common Criteria
- Correctness of the generated code, \( \forall \) alea
Génération de code au runtime pour la sécurité des systèmes embarqués

damien.courrousse@cea.fr
APPLICATION TO AES: ADDROUNDKEY()

void addRoundKey_compilette( cdg_insn_t* code
       , uint8_t* key_addr, uint8_t *state_addr)
{
    #[ Begin code Prelude
    Type reg32 int 32
    Alloc reg32 state, key, i
    mv i, #(16)
    loop:
        sub i, i, #(1)
        lb state, @(#(state_addr) + i) // state = state_addr[i]
        lb key, @(#(key_addr) + i)      // key= key_addr[i]
        xor state, key
        sb @(#(state_addr) + i), state
        bneq loop, i, #(0)
    rtn
    End
    ]#
}

Same code for performance purposes and for polymorphism.
The security protections are added in the backend.
**APPROACHES FOR CODE SPECIALISATION**

**Static code versionning** (e.g. C++ Templates)

- **source code** → **compiler** → **IR** → **executable** → **RUNTIME**

**Runtime code generation**, with deGoal

A *compilette* is an ad hoc code generator, targeting one executable

- **source code** → **compiler** → **IR** → **compilette** → **executable** → **RUNTIME**

**Dynamic compilation** (JITs, e.g. Java Hotspot)

- **source code** → **compiler** → **IR** → **bytecode** → **JIT** → **executable** → **RUNTIME**

**Intermediate Representation (IR)**

- **static compilation**
- **runtime: select executable**
- **memory footprint ++**
- **limited genericity**
- **runtime blindness**

- **fast code generation**
- **memory footprint ---**
- **data-driven code generation**

- **overhead ++**
- **memory footprint ++**
- **not designed for data dependant code-optimisations**
## DEGOAL SUPPORTED ARCHITECTURES

<table>
<thead>
<tr>
<th>ARCHITECTURE</th>
<th>STATUS</th>
<th>FEATURES</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARM32</td>
<td>✔</td>
<td></td>
</tr>
<tr>
<td>ARM Cortex-A, Cortex-M [Thumb-2, VFP, NEON]</td>
<td>✔</td>
<td>SIMD, [IO/OoO]</td>
</tr>
<tr>
<td>STxP70 [including FPx] (STHORM / P2012)</td>
<td>✔</td>
<td>SIMD, VLIW (2-way)</td>
</tr>
<tr>
<td>K1 (Kalray MPPA)</td>
<td>✔</td>
<td>SIMD, VLIW (5-way)</td>
</tr>
<tr>
<td>PTX (Nvidia GPUs)</td>
<td>✔</td>
<td></td>
</tr>
<tr>
<td>MIPS</td>
<td>⌚</td>
<td>32-bits</td>
</tr>
<tr>
<td>MSP430 (TI microcontroller)</td>
<td>✔</td>
<td>Up to &lt; 1kB RAM</td>
</tr>
</tbody>
</table>

**CROSS CODE GENERATION supported**
(e.g. generate code for STxP70 from an ARM Cortex-A)

[IO/OoO]: Instruction scheduling for in-order and out-of-order cores
RANDOM REGISTER ALLOCATION

- Greedy algorithm: each register is allocated among one of the free registers remaining

- Has an impact on:
  - The management of the context (ABI)
  - Instruction selection
Replace an instruction by a semantically equivalent sequence of one or several instructions

Select the sequence in a list of equivalences

Examples:

\[
\begin{align*}
c & := a \text{ xor } b \iff c := ((a \text{ xor } r) \text{ xor } b) \text{ xor } r \\
c & := a \text{ xor } b \iff c := (a \text{ or } b) \text{ xor } (a \text{ and } b) \\
c & := a - b \iff k := 1 \; ; \; c := (a + k) + (\text{not } b) \\
c & := a - b \iff c := ((a + r) - b) - r
\end{align*}
\]
Reorder instructions

... but do not break the semantics of the code!

- Defs – read registers
- Uses – modified registers
- *Do not* take into account result latency and issue latency
- Special treatments for... special instructions. E.g. branch instructions
Noise instructions have no effect on the result of the program

- Parametrable model of the inserted delay ~ program execution time
  - Goal
    - Maximum standard deviation $\sigma$
    - Minimum mean $E$
  - Can insert any instruction:
    - nop
    - Arithmetic (add, xor...)
    - Memory accesses (lw, lb, ...)
    - Power hungry instructions (mul, mac...)

$N$: number of insertions

$(E, \sigma) = f(N)$

- the noise model
- the generated code
Demo

8-bit AES
STM32 (Cortex-M3)

polymorphic version
Lib. Polymorphic code generation
polymorphic AES 8 bits
polymorphic instances of AES

reference version
AES 8 bits.c

leti & list