Runtime Code Polymorphism as a Protection against Physical Attacks
Damien Couroussé, Bruno Robisson, Thierno Barry, P Jaillon, Olivier Potin

To cite this version:

HAL Id: emse-01232662
https://hal-emse.ccsd.cnrs.fr/emse-01232662
Submitted on 23 Nov 2015

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
Core Idea: Runtime Code Polymorphism

Definition
Regularly changing the behaviour of a (secured) component, at runtime, while maintaining unchanged its functional properties.

What for?
- Protection against reverse engineering of SW
- the secured code is not available before runtime
- the secured code regularly changes its form (code generation interval \( \omega \geq 1 \))
- Protection against physical attacks
- polymorphism changes the spatial and temporal properties of the secured code: side channel & fault attacks
- combine with usual SW protections against focused attacks

How?
- deGoal: runtime code generation for embedded systems
- fast code generation
- tiny memory footprint: proof of concept on TI's MSP430 (512 bytes of RAM)

Compiettes & deGoal in a Nutshell

Polymorphic Code Generation

deGoal runtime capabilities
Performed in this order:
- register selection
- instruction selection
- instruction scheduling

Adaptation to achieve runtime code polymorphism:
- Portability to very small processors and secure elements
  - Limited memory consumption
  - Fast runtime code generation
- Ability to combine with hardware countermeasures
- Introduce alea during runtime code generation [1,2,3]
- Polymorphism:
  - random mapping to physical registers [1]
  - use of semantic equivalences [2]
  - instruction scheduling [3]
  - insertion of dummy operations [3]

Example: polymorphic AES

Polymorphic implementation of the SubBytes function:
```c
void gen_subBytes( cdg_insn_t* code , uint8_t* state_addr , uint8_t* sbox_addr)
{
    #
    Begin code Prelude
    Type uint32 state, sbox, i, x, y
    nv state, #(@state_addr)
    nv sbox, #(@sbox_addr)
    nv i, #(0)
    loop:
        lb x, #(@state+i) // x := state[i]
        lb y, #(@sbox+x) // y := sbox[x]
        sb @(@state+i), y // state[i] := y
        add i, i, #(1)
        bneq loop, i, #(16)
    rt nb
    End
}
```

Execution times (in cycles), over 1000 runs:

<table>
<thead>
<tr>
<th></th>
<th>min</th>
<th>max</th>
<th>average</th>
</tr>
</thead>
<tbody>
<tr>
<td>reference</td>
<td>6385</td>
<td>6385</td>
<td>6385</td>
</tr>
<tr>
<td>code generator</td>
<td>5671</td>
<td>12910</td>
<td>9345</td>
</tr>
<tr>
<td>polymorphic inst</td>
<td>7185</td>
<td>9745</td>
<td>8303</td>
</tr>
</tbody>
</table>

Impact of the code generation interval \( \omega \):

<table>
<thead>
<tr>
<th>( \omega )</th>
<th>k</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2.76</td>
<td>53.0%</td>
</tr>
<tr>
<td>5</td>
<td>1.19</td>
<td>18.4%</td>
</tr>
<tr>
<td>20</td>
<td>1.37</td>
<td>2.1%</td>
</tr>
<tr>
<td>100</td>
<td>1.31</td>
<td>1.1%</td>
</tr>
</tbody>
</table>

k: overhead vs. reference implementation
%: percentage contribution of runtime code generation to the performance overhead

References