Compilation for cyber-security in embedded systems

Workshop SERTIF

Damien Couroussé
CEA – LIST / LIALP; Grenoble Université Alpes
damien.courousse@cea.fr
Typology of attacks

- Cryptanalysis
  Out of our scope

- **Passive attacks. side channel attacks**
  Observations: power, electro-magnetic, execution time, temperature, etc.

- **Active attacks. fault attacks**
  Over/under voltage, laser, ion beam, EM, clock glitches...

- **Reverse engineering**
  Hardware inspection: mechanical or chemical etching, scope observation...
  Software inspection: debug, memory dumps, code analysis...
  Using physical attacks: SCARE, FIRE...

- **Logical attacks**
  Not (yet) considered

Real world attacks are different from research literature

- **First step.** Global analysis, calibration of the attack bench(es), identification of weaknesses

- **Second step.** The textbook attack: a focused attack on a known weakness
FROM THE IDEA TO A PRODUCT: A DETAILED PLAN

implementation

source code

compilation

machine code

Static code securisation: compilation

Dynamic code securisation: polymorphism
STATIC COMPILATION OF PROTECTIONS AGAINST FAULT ATTACKS
AUTOMATED APPLICATION OF COUNTERMEASURES WITH STATIC COMPILATION

- Access to program’s semantics (e.g. secret variable)
- Security properties are not guaranteed, post compilation
- Corollary: can lead to bigger overheads

- Access to program semantics
- Control over machine code
- Benefit from compiler optimisations
- Implementation within the compiler is difficult

- Naturally fits to low-level / machine code protection schemes
- (Re-)construction of a program representation is difficult
- Mostly ad hoc protection schemes

Source code

Source to source approach

Compiler

Assembly approach

Binary code
COMPILATION OF A COUNTERMEASURE AGAINST INSTRUCTION SKIP FAULT ATTACKS

- Attack model: faults, instruction skip
- Protection model: instruction redundancy
  - Formally verified countermeasure model
    [Moro et al., 2014]

Platform

- STM32 F100: ARM Cortex-M3
- Frequency: 24 MHZ
- Instruction Set: Thumb2

Workflow

Source Code → LLVM Compiler + CounterMeasure → ARM assembly Code → ARM Cross Compiler → Binary code → STM32 F100
“An instruction is idempotent when it can be re-executed several times with always the same result”

**Example**

**is idempotent**

- **add** R0, R1, R2

  **Duplication**
  - add R0, R1, R2
  - add R0, R1, R2

**Is not idempotent**

- **add** R1, R1, R2
  - Transformation **[Moro et al. 2014]**
  - add R1, R1, R2
  - add R1, R1, R2

**Issues**

- How to find a free register at the assembly code level?
  - **[Barenghi et al. 2010]**: ad hoc implementation. The number of free registers is known for their implemented AES
  - **[Moro et al. 2014]**: use of the scratch register r12

- **Overhead:**
  - At least $\times 4$ for each instruction
  - **[Moro et al. 2014]** Reported $\times 14$ for the ARM instruction: `umlal`
INSTRUCTION DUPLICATION WITH LLVM

- Instruction selection
  - Force three-operands instructions
- Register allocation
  - Force the use of different registers for source and destination operands
    \[ \text{add } r0, r0, r1 \Rightarrow \text{add } r2, r0, r1 \]
- Transformation passes
  - Transformation of non-idempotent instructions into a sequence of idempotent ones
- Instruction duplication
  - Straightforward. Could (should?) be executed after instruction scheduling
COMPILATION OF A COUNTERMEASURE AGAINST INSTRUCTION SKIP FAULT ATTACKS

- Attack model: faults, instruction skip
- Protection model: instruction redundancy
  - Formally verified countermeasure model [Moro et al., 2014]

Platform
STM32 F100: ARM Cortex-M3
Frequency: 24 MHz
Instruction Set: Thumb2

Experimental results for AES

<table>
<thead>
<tr>
<th>Opt. level</th>
<th>Moro et al.’s AES</th>
<th>MiBench AES</th>
<th>Overhead</th>
<th>Moro et al [3]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Unprotected</td>
<td>Protected</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>exec. time</td>
<td>size</td>
<td>exec. time</td>
<td>size</td>
</tr>
<tr>
<td>-O0</td>
<td>17940</td>
<td>1736</td>
<td>29796</td>
<td>3960</td>
</tr>
<tr>
<td>-O1</td>
<td>9814</td>
<td>1296</td>
<td>18922</td>
<td>2936</td>
</tr>
<tr>
<td>-O2</td>
<td>5256</td>
<td>1936</td>
<td>9934</td>
<td>4184</td>
</tr>
<tr>
<td>-O3</td>
<td>5256</td>
<td>1936</td>
<td>9934</td>
<td>4184</td>
</tr>
<tr>
<td>-Os</td>
<td>7969</td>
<td>1388</td>
<td>16084</td>
<td>3070</td>
</tr>
</tbody>
</table>

DYNAMIC PROTECTION:
CODE POLYMORPHISM
Definition

Regularly **changing the behavior of a (secured) component, at runtime, while maintaining unchanged its functional properties**, with 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 physical attacks: side channel & fault attacks

- Polymorphism changes the spatial and temporal properties of the secured code.
- Compatible with State-of-the-Art HW & SW Countermeasures.

Protection against reverse engineering of SW

- The secured code is not available before runtime.
- The secured code regularly changes its form (code generation interval $\omega$).

DeGoal: runtime code generation for embedded systems

- Fast code generation, tiny memory footprint.
AES, 8-bit STM32 (Cortex-M3)
IMPACT OF POLYMORPHISM ON 1ST ORDER CPA

Reference version: unprotected AES-8
IMPACT OF POLYMORPHISM ON CPA

Effect of the code generation interval

Reference implementation

- Distinguish threshold = 39 traces

Polymorphic version, code generation intervall: 500

- Distinguish threshold = 89 traces
IMPACT OF POLYMORPHISM ON CPA

Polymorphic version
code generation interval: **20**

![Graph showing peak correlation coefficient vs. number of traces.](image)

Distinguish threshold > 10000 traces

---

Polymorphic version,
code generation interval: **500**

![Graph showing peak correlation coefficient vs. number of traces.](image)

Distinguish threshold = 89 traces

---
Reference version:

foo.c
AES 8 bits.c

Polymorphic version, with COGITO:

foo.c
AES.cdg

COGITO
AES.cdg.c

Platform compiler
Polymorphic code generation lib.
Polymorphic code generator

Binary image
rand()
AES SubBytes: polymorphic loop

```c
void subBytes_compilette(cdg_insn_t* code, const byte* sbox_addr, unsigned char* state_addr)
{
    #[Begin code Prelude
    Type uint32 int 32
    Alloc uint32 rstate, rstatei, rsbox, rsboxi, i
    mv rsbox, #((unsigned int)sbox_addr)
    noise_load_setup rsbox, #(256)
    mv rstate, #((unsigned int)state_addr)
    ]#
    /* insert [0; 32[ noise instructions */
    cdg_gennoise_((((PRELUDE_NOISE_LEVEL - 1) << 4) & cdg_rand()) >> 4);
    #[
    mv i, #(0)
    loop:
        lb rstatei, rstate, i //statei = state[i]
        lb rsboxi, rsbox, rstatei //sboxi = sbox[statei]
        sb rstate, i, rsboxi //state[i] = sboxi
        add i, i, #(1)
        bneq loop, i, #(16)
    rtn
    End
    ]#;
}
```

Side channel leakage of key data
## SUBBYTES: SAMPLES OF GENERATED MACHINE CODE

### Reference version

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Reference</th>
<th>Polymorphic instances</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>stmdb</code></td>
<td>sp!, {r4, r7}</td>
<td><code>stmdb</code> sp!, {r5, r6, r7, r8, r9, r10, r11, sp}, {r4, r5, r6, r7, r10, r11, lr} sp!, {r4, r5, r6, r7, sp}, r9, lr</td>
</tr>
<tr>
<td><code>movw</code></td>
<td>r2, #33380</td>
<td><code>movw</code> r12, #58220, <code>movw</code> r12, #58220, <code>movw</code> r12, #58220, <code>movw</code> r12, #58220, <code>movw</code> r12, #58220, <code>movw</code> lr, #58220</td>
</tr>
<tr>
<td><code>movt</code></td>
<td>r2, #2049</td>
<td><code>movt</code> r12, #2049, <code>movt</code> r12, #2049, <code>movt</code> r12, #2049, <code>movt</code> r12, #2049, <code>movt</code> r12, #2049, <code>movt</code> lr, #2049</td>
</tr>
<tr>
<td><code>movw</code></td>
<td>r0, #44</td>
<td><code>movw</code> r2, #64, <code>movw</code> r2, #64, <code>movw</code> lr, #0, <code>subs</code> r2, r2, r2</td>
</tr>
<tr>
<td><code>movt</code></td>
<td>r0, #8192</td>
<td><code>movt</code> r3, r3, r3, <code>movt</code> r3, r3, r3, <code>movt</code> r3, r3, r3, <code>movt</code> r3, r3, r3, <code>movt</code> r3, r3, r3, <code>movt</code> r5, #64</td>
</tr>
<tr>
<td><code>subs</code></td>
<td>r3, r3, r3</td>
<td><code>subs</code> r0, r0, r0, <code>subs</code> r0, r0, r0, <code>subs</code> r0, r0, r0, <code>subs</code> r0, r0, r0, <code>subs</code> r0, r0, r0, <code>subs</code> r8, #64</td>
</tr>
<tr>
<td><code>addw</code></td>
<td>r4, r4, #1</td>
<td><code>addw</code> r10, r10, r10, <code>addw</code> r10, r10, r10, <code>addw</code> r10, r10, r10, <code>addw</code> r10, r10, r10, <code>addw</code> r10, r10, r10, <code>addw</code> r10, r10, r10</td>
</tr>
<tr>
<td><code>cmp</code></td>
<td>r4, #16</td>
<td><code>cmp</code> r9, #0, <code>cmp</code> r9, #0, <code>cmp</code> r9, #0, <code>cmp</code> r9, #0, <code>cmp</code> r9, #0, <code>cmp</code> r9, #0</td>
</tr>
<tr>
<td><code>bne.n</code></td>
<td><code>ldmdb</code></td>
<td><code>bne.n</code> <code>0x20000852</code></td>
</tr>
<tr>
<td><code>ldmia.w</code></td>
<td>sp!, {r4, r7}</td>
<td><code>ldmia.w</code> sp!, {r5, r6, r7, r8, r9, r10, r11, lr} sp!, {r4, r5, r6, r7, sp}, r9, lr</td>
</tr>
<tr>
<td><code>bx</code></td>
<td>lr</td>
<td><code>bx</code> lr</td>
</tr>
</tbody>
</table>

**Random register allocation**

**Instruction substitution**

**Insertion of noise instructions**

**Instruction shuffling**

**No code suppression**
AES SubBytes: polymorphic loop

```c
void subBytes_compilette(cdg_insn_t* code, const byte* sbox_addr,
                          unsigned char* state_addr)
{
    #[Begin code Prelude
    Type uint32 int 32
    Alloc uint32 rstate, rstatei, rsbox, rsboxi, i
    mv rsbox, #((unsigned int)sbox_addr)
    noise_load_setup rsbox, #(256)
    mv rstate, #((unsigned int)state_addr)
    ]#
    /* insert [0; 32[ noise instructions */
    cdg_gennoise_(((PRELUDE_NOISE_LEVEL - 1) << 4) & cdg)
    #[
    mv i, #(0)
    loop:
        lb rstatei, rstate, i //statei = state[i]
        lb rsboxi, rsbox, rstatei //sboxi = sbox[sta
        sb rstate, i, rsboxi //state[i] = sboxi
        add i, i, #(1)
        bneq loop, i, #(16)
    rtn
    End
    ]#;
}
```
TIME DISPERSION OF THE LEAKAGE POINT

AES SubBytes: polymorphic loop + shuffling

```c
void subBytes_compilette(cdg_insn_t* code, const byte* sbox_addr, unsigned char* state_addr)
{
    #[
    Begin code Prelude
    Type uint32 int 32
    Alloc uint32 rstate, rstatei, rsbox, rsboxi, i
    mv rsbox, #((unsigned int)sbox_addr)
    noise_load_setup rsbox, #(256)
    mv rstate, #((unsigned int)state_addr)
    ]#
    /* insert [0; 32[ noise instructions */
    cdg_gennoise_(((PRELUDE_NOISE_LEVEL - 1) << 4) & cdg_rand()) >> 4);
    int indices[SBOX_INDICES_LEN];
    init_and_permute_table(indices, SBOX_INDICES_LEN);
    for(i=0; i<16; i++) {
    #[
    Alloc uint32 rstatei, rsboxi
    lb rsboxi, rsbox, rstatei //sboxi = sbox[statei]
    sb rstate, #(indices[i]), rsboxi //state[i] = sboxi
    Free rstatei, rsboxi
    ]#
    }
}
```
# SUBBYTES:
## SAMPLES OF GENERATED MACHINE CODE

<table>
<thead>
<tr>
<th>Reference version</th>
<th>New reference version, unrolled</th>
<th>Polymorphic instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>stmdb</td>
<td>sp!, {r4, r7}</td>
<td>stmdb</td>
</tr>
<tr>
<td>movw</td>
<td>r2, #33380</td>
<td>movw</td>
</tr>
<tr>
<td>movt</td>
<td>r2, #2049</td>
<td>movt</td>
</tr>
<tr>
<td>movw</td>
<td>r0, #44</td>
<td>movw</td>
</tr>
<tr>
<td>movt</td>
<td>r0, #8192</td>
<td>movt</td>
</tr>
<tr>
<td>movs</td>
<td>r4, #0</td>
<td>ldrb</td>
</tr>
<tr>
<td>ldrb</td>
<td>r1, [r0, r4]</td>
<td>ldrb</td>
</tr>
<tr>
<td>strb</td>
<td>r3, [r0, r4]</td>
<td>strb</td>
</tr>
<tr>
<td>addw</td>
<td>r4, r4, #1</td>
<td>ldrb</td>
</tr>
<tr>
<td>cmp</td>
<td>r4, #16</td>
<td>strb</td>
</tr>
<tr>
<td>bne.n</td>
<td>0x20000852</td>
<td>ldrb</td>
</tr>
<tr>
<td>ldmia.w</td>
<td>sp!, {r4, r7}</td>
<td>ldrb</td>
</tr>
<tr>
<td>bx</td>
<td>lr</td>
<td>strb</td>
</tr>
</tbody>
</table>

- **stmdb** sp!, {r4, r7}
- **movw** r2, #33380
- **movt** r2, #2049
- **movw** r0, #44
- **movt** r0, #8192
- **movs** r4, #0
- **ldrb** r1, [r0, r4]
- **strb** r3, [r0, r4]
- **addw** r4, r4, #1
- **cmp** r4, #16
- **bne.n** 0x20000852
- **ldmia.w** sp!, {r4, r7}
- **bx** lr

- **stmdb** sp!, {r7}
- **movw** r1, #7723
- **movt** r1, #2048
- **movw** r0, #44
- **movt** r0, #8192
- **ldrb** r3, [r0, #0]
- **ldrb** r2, [r1, r3]
- **strb** r2, [r0, #0]
- **ldrb** r3, [r0, #1]
- **ldrb** r2, [r1, r3]
- **strb** r2, [r0, #1]
- **ldrb** r3, [r0, #2]
- **ldrb** r2, [r1, r3]
- **strb** r2, [r0, #2]
- **ldrb** r3, [r0, #3]
- **ldrb** r2, [r1, r3]
- **strb** r2, [r0, #3]
- **ldrb** r3, [r0, #4]
- **ldrb** r2, [r1, r3]
- **strb** r2, [r0, #4]
- **ldrb** r3, [r0, #5]
- **ldrb** r2, [r1, r3]
- **strb** r2, [r0, #5]
- **ldrb** r3, [r0, #6]
- **ldrb** r2, [r1, r3]
- **strb** r2, [r0, #6]
- **ldrb** r3, [r0, #7]
- **ldrb** r2, [r1, r3]
- **strb** r2, [r0, #7]
- **ldrb** r3, #0
- **ldrb** r2, [r1, r3]
- **strb** r2, [r0, #8]
- **ldrb** r3, #1
- **ldrb** r2, [r1, r3]
- **strb** r2, [r0, #9]
- **ldrb** r3, #2
- **ldrb** r2, [r1, r3]
- **strb** r2, [r0, #10]
- **ldrb** r3, #3
- **ldrb** r2, [r1, r3]
- **strb** r2, [r0, #11]
- **ldrb** r3, #4
- **ldrb** r2, [r1, r3]
- **strb** r2, [r0, #12]
- **ldrb** r3, #5
- **ldrb** r2, [r1, r3]
- **strb** r2, [r0, #13]
- **ldrb** r3, #6
- **ldrb** r2, [r1, r3]
- **strb** r2, [r0, #14]
- **ldrb** r3, #7
- **ldrb** r2, [r1, r3]
- **strb** r2, [r0, #15]
- **ldrb** r3, #8
- **ldrb** r2, [r1, r3]
- **strb** r2, [r0, #16]
AES SubBytes: polymorphic loop + shuffling

```c
void subBytes_compilette(cdg_insn_t* code, const byte* sbox_addr, unsigned char* state_addr)
{
    #[Begin code Prelude
    Type uint32 int 32
    Alloc uint32 rstate, rstatei, rsbox, rsboxi, i
    mv rsbox, #((unsigned int)sbox_addr)
    noise_load_setup rsbox, #(256)
    mv rstate, #((unsigned int)state_addr)
    ]#
    /* insert [0; 32[ noise instructions */
    cdg_gennoise_(((PRELUDE_NOISE_LEVEL - 1) << 4) & cdg_int indices[SBOX_INDICES_LEN];
    init_and_permute_table(indices, SBOX_INDICES_LEN);
    for(i=0; i<16; i++) {
        #[Alloc uint32 rstatei, rsboxi
        lb rsboxi, rsbox, rstatei       //sboxi = sbox
        sb rstate, #(indices[i]), rsboxi    //state[i] = s
        Free rstatei, rsboxi
        ]#
    }
}
```
Polymorphism is a *hiding* countermeasure

- The leakage instruction:
  - is not modified as compared to the same instruction in the reference version (for eval purposes)
- Unprotected AES: $N < 50$ traces $\Rightarrow$ Polymorphic: $N > 1000000$ traces
  - But data leakage is still available in the traces

... compatible with masking
Compilation for cyber-security in embedded systems

Damien Couroussé
CEA – LIST / LIALP; Grenoble Université Alpes
damien.courousse@cea.fr