SECURING EMBEDDED SOFTWARE WITH COMPILERS

Damien Couroussé | CEA / LIST / DACLE
CEA-LETI LID, Minatec Grenoble, 2019-06-28
A major threat against secure embedded systems

• The most effective attacks against implementations of cryptography
• Relevant against many parts of CPS/IoT: bootloaders, firmware upgrade, etc.
• Recently used to leverage software vulnerabilities [1]

In practice,

• An attacker mostly uses logical attacks if the target is unprotected (e.g. typical IoT devices): buffer overflows, ROP, protocol vulnerabilities, etc.
• All high security products embed countermeasures against side-channel and fault injection attacks. E.g. Smart Cards, payTV, military-grade devices.
  • Using a combination of hardware and software countermeasures
  • Tools for Side-channel and fault injection are getting really affordable

AUTOMATED APPLICATION OF COUNTERMEASURES WITH A COMPILER

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

**Source to source approach**

**Compiler**

**Assembly approach**
- ✔ Access to program semantics
- ✔ Control over machine code
- ✔ Benefit from compiler optimisations
- ✗ Implementation within the compiler is difficult
- ✗/✔ Focus on generic countermeasures

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

← our approach
Automated application of software countermeasures against physical attacks

⇒ A toolchain for the compilation of secured programs

Several countermeasures
- **Fault tolerance**, including multiple fault injections
- **Execution Integrity & Control-Flow Integrity**
  - Detection of perturbations on the instruction path, at the granularity of a single machine instruction
- **Side channel hiding**

Tools for **security and performance evaluations**

Based on **LLVM**: an industry-grade, state-of-the-art compiler (competitive with GCC)
SECURING AND VERIFYING PROGRAMS

- **Compilation**: automation of the application of software countermeasures against fault attacks and side-channel attacks
- **Functional verification**: of the secured machine code (equivalence with an unprotected version of the same program)
- **Security verification**: correctness of the applied countermeasures w.r.t a security model

On-going joint work with LIP6, Paris (PROSECCO – ANR 2015)
Objective: the program is not perturbed by the injection of faults

- Countermeasure based on a protection scheme **formally verified for the ARM architecture** [Moro et al., 2014, Barry et al. 2016]
- **Automatic application** by the compiler
- Allow to **parameterize** level of protection
- **Generalisation** of [Moro et al., 2014] to **multiple faults** of **configurable width**
- Target: ARM Cortex-M cores

- **Fine-grained countermeasure** applied to critical functions **reduces the execution overhead** below x1.23 and **size overheads** below x1.12 [Barrys’ thesis, 2017]

---


Objective: monitoring program execution integrity, at runtime

Combined protections:
- Protection of the control-flow of an application (Control-Flow Integrity)
- Beyond CFI: protection of branchless sequences of instructions, at the granularity of a single machine instruction

Coverage:
- Alteration of the PC (instruction skips, branches)
- Corruption of branches
- Alteration of branch conditions

Two implementations
- Software only countermeasure. Implementation for ARM
- HW-SW countermeasure. Fine-grain execution integrity, verification & authentication.
Code polymorphism: regularly changing the observable behavior of a program, at runtime, while maintaining unchanged its functional properties,

- Protection against physical attacks: side channel & fault attacks
  - Changes the spatial and temporal properties of the secured code
  - Can be combined with other state-of-the-Art HW & SW Countermeasures
- Can run on low-end embedded systems with only a few kB of memory
  - Illustrated below: STM32F1 microcontroller with 8kB of RAM

Compliant with certification standards (Common Criteria, CSPS, etc.)
CODE POLYMORPHISM: WORKING PRINCIPLE

Runtime code generation for embedded systems

Reference version:

foo.c
AES.c

Polymorphic version, with COGITO:

foo.c
AES.c

COGITO compiler

AES.odo.c
Polymorphic code generation lib.

Platform compiler

Binary image
AES

Platform compiler

Binary image
Polymorphic AES code generator

rand()
<table>
<thead>
<tr>
<th>CODE TRANSFORMATIONS USED AT RUNTIME</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Register shuffling</strong></td>
</tr>
<tr>
<td><strong>RANDOM</strong> general purpose</td>
</tr>
<tr>
<td>register permutation</td>
</tr>
<tr>
<td>add r4, r4, r5</td>
</tr>
<tr>
<td>xor r6, r5, r8</td>
</tr>
<tr>
<td>add r11, r11, r7</td>
</tr>
<tr>
<td>xor r8, r7, r5</td>
</tr>
<tr>
<td><strong>Instruction shuffling</strong></td>
</tr>
<tr>
<td>independent instructions are emitted</td>
</tr>
<tr>
<td>in a <strong>RANDOM</strong> order</td>
</tr>
<tr>
<td>xor r6, r5, r8</td>
</tr>
<tr>
<td>add r4, r4, r5</td>
</tr>
<tr>
<td>Semantic variants</td>
</tr>
<tr>
<td>replacement of an instruction by a</td>
</tr>
<tr>
<td><strong>RANDOMLY</strong> selected semantic variant</td>
</tr>
<tr>
<td>add r4, r4, r5</td>
</tr>
<tr>
<td>xor r6, r5, #12348</td>
</tr>
<tr>
<td>xor r6, r6, r8</td>
</tr>
<tr>
<td>xor r6, r6, #12348</td>
</tr>
<tr>
<td><strong>Noise instructions</strong></td>
</tr>
<tr>
<td>insertion of a <strong>RANDOM</strong> number of</td>
</tr>
<tr>
<td><strong>RANDOMLY</strong> chosen noise instructions</td>
</tr>
<tr>
<td>add r4, r4, r5</td>
</tr>
<tr>
<td>sub r7, r6, r2</td>
</tr>
<tr>
<td>load r3, r10, #53</td>
</tr>
<tr>
<td>xor r6, r5, r8</td>
</tr>
<tr>
<td><strong>Dynamic noise</strong></td>
</tr>
<tr>
<td><strong>RANDOM</strong> insertion of noise</td>
</tr>
<tr>
<td>instructions with a <strong>RANDOM</strong> jump</td>
</tr>
<tr>
<td>add r4, r4, r5</td>
</tr>
<tr>
<td>jump 0, 1 or 2 instructions</td>
</tr>
<tr>
<td>sub r7, r6, r2</td>
</tr>
<tr>
<td>load r3, r10, #53</td>
</tr>
<tr>
<td>xor r6, r5, r8</td>
</tr>
</tbody>
</table>
### TOOLCHAIN CONFIGURABILITY

<table>
<thead>
<tr>
<th>Period of regeneration</th>
<th>Register shuffling</th>
<th>Instruction shuffling</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\mathbb{N}$</td>
<td>${0, 1}$</td>
<td>${0, 1}$</td>
</tr>
<tr>
<td>(or custom regeneration policies)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Total configuration space:**

$$\{0, 1\}^2 \times \{0, 1, 2\}^2 \times \mathbb{R} \times \mathbb{N}^3$$

<table>
<thead>
<tr>
<th>Semantic variants</th>
<th>Noise instructions</th>
<th>Dynamic noise</th>
</tr>
</thead>
<tbody>
<tr>
<td>${0, 1, 2}$</td>
<td>${0, 1, 2} \times \mathbb{R} \times \mathbb{N}$</td>
<td>$\mathbb{N}$</td>
</tr>
</tbody>
</table>

**A huge number of polymorphic variants:**

- 10 original machine instructions $\Rightarrow 6 \times 10^{42}$ variants
- AES with 278 machine instructions $\Rightarrow 10^{27}$ variants (pessimist bound)
• Basis: polymorphic configuration with low variability
• Acquisition of traces from Electro-Magnetic observations
• CPA on SBOX 1st output with HW model
• Experimental platform not designed for security applications (hence the weak results on the unprotected version)

Experimental results
• This polymorphic version requires 13000x more traces
• Execution time overhead: x2.5 including generation cost

More results in [TACO 2019]
AUTOMATED APPLICATION OF CODE POLYMORPHISM

Declaration of polymorphism with a compiler option
- polymorphic-function foo will compile function foo into a polymorphic implementation,
- polymorphic will compile all functions found in the compiled source into polymorphic implementations.

Many configurable levels of polymorphic transformations => security/performance tradeoff
- Nature and parameters of the code transformations: random allocation of registers, semantic variants, instruction shuffling, insertion of noise instructions.
- Frequency and policy for runtime code regeneration
- Memory protections
- Leveraging OS-level features, e.g. concurrency

Components evaluated: ciphers, hash functions, simple authentication, random generated codes (Csmith*)

![Graph showing execution time overhead and size overhead for various components and polymorphism levels.](image)
## PERFORMANCE EVALUATION
OF RUNTIME CODE GENERATION

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Execution time overhead (geo. mean)</th>
<th>Size overhead (geo. mean)</th>
</tr>
</thead>
<tbody>
<tr>
<td>none</td>
<td>x1.40</td>
<td>x1.70</td>
</tr>
<tr>
<td>low</td>
<td>x2.31</td>
<td>x2.87</td>
</tr>
<tr>
<td>medium</td>
<td>x2.45</td>
<td>x3.44</td>
</tr>
<tr>
<td>high</td>
<td>x4.03</td>
<td>x3.81</td>
</tr>
</tbody>
</table>

Overheads depend on configuration → trade-off to find

Runtime code generation done in linear complexity

More results in [TACO 2019]
Bottlenecks for the use of runtime code generation in embedded systems:

- Memory allocation of code buffers
  - No Operating System (no `malloc`), no virtual memory.

- Management of memory permissions (read, write, execute)
  - Runtime code generation requires write access to program memory

Polymorphic version, with COGITO:

- `foo.c`
- `AES.c`
- `AES.odo.c`
- `Platform compiler`
- `Binary image`
  - Polymorphic AES code generator
- `rand()`
Challenges

- No Operating System, no dynamic memory allocation (malloc), no MPU
- Generated code has a variable size
- Largest possible code size does not fit in system memory

Idea: compute a realistic code size suitable for \((1-p)\) code generations.

- Threshold \(p\): probability of memory overflow
- \(p = 10^{-6}\) by default
- Computation of the code size done automatically by the compiler

For a 100 instructions code (low config.), allocated size is 5x smaller than worst case!
PREVENTION OF CODE BUFFER OVERFLOWS

STATICALLY

1. COGITO compiler
2. Platform compiler
3. binary

RUNTIME

1. computes the size of useful instructions
2. puts the information directly in runtime code generator’s code
3. always keep space for useful instructions (limit polymorphism if necessary)
**Objective:** Guarantee $W \oplus X$ and that only the generator can write into the buffer

<table>
<thead>
<tr>
<th>Specialized runtime code generator</th>
<th>Interrupt handler</th>
<th>Instance buffer (memory allocated)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Generation begins</td>
<td>raise interrupt</td>
<td>X only</td>
</tr>
<tr>
<td>ERROR</td>
<td></td>
<td>W only</td>
</tr>
<tr>
<td>Emit code</td>
<td>Check address of interrupt</td>
<td></td>
</tr>
<tr>
<td>ERROR</td>
<td>bad</td>
<td>W only</td>
</tr>
<tr>
<td>resume execution</td>
<td>good</td>
<td>X only</td>
</tr>
<tr>
<td>Generation ends</td>
<td>Check address of interrupt</td>
<td></td>
</tr>
<tr>
<td>ERROR</td>
<td>bad</td>
<td>W only</td>
</tr>
<tr>
<td>resume execution</td>
<td>good</td>
<td>X only</td>
</tr>
<tr>
<td>raise interrupt</td>
<td></td>
<td>W only</td>
</tr>
</tbody>
</table>
• Leverage the compiler to implement counter-measures
  • Automation, flexibility, configurability
• Leverage compiler analysis and compiler optimisations to improve the effectiveness of counter-measures

Ongoing directions
• Hardware security with software-only counter-measures is impossible challenging
  • Challenge your threat model
  • HW/SW co-design of countermeasures