Automated combination of tolerance and control flow integrity countermeasures against multiple fault attacks on embedded systems

Thierno Barry
PhD Candidate in security of Embedded Systems
at CEA (the French Atomic Energy Commission)
thierno.barry@cea.fr

Damien Couroussé (CEA)  Karine Heydemann (LIP6)  Bruno Robisson (CEA)

2017 European LLVM Developers’ Meeting
March 28, 2017, Saarbrücken, Germany
• Embedded systems have increasingly become critical part of our daily life

• One of the major threats against these systems are physical attacks
  
  Side Channel Attacks

  Fault Injection Attacks

  • These attacks essentially aim to:

  Obtain sensitive data  Bypass protections  Reverse engineering

• The security of these systems reveals itself as major concern for both industrials and state organizations
Our work consists in generating codes that are protected against these attacks
• A number of software-based countermeasures against fault attacks already exist

• Security properties cannot be guaranteed after code compilation [Balakrishan et al. 2008]

• Except if the compiler code optimizers are disabled as suggested in [Eldib et al. 2014]
  leads to a very high overheads +400% in [Lalande et al. 2014]

• Unlike the source to source approach we have control over code optimizers

• Unlike assembly approach we have the benefit of code transformation opportunities provided by the compiler
  Allows to reduce the security overhead

• Lack of semantic information

• Several transformations need to be performed
  leads to significant overheads [Moro et al. 2014]
• Each countermeasure is designed to protect against one single attack

\[ \text{Attack} \rightarrow \text{Countermeasure} \]

• When it comes to protect against several attacks:

\[ \text{3 Attacks} \longrightarrow \text{Countermeasure} \longrightarrow \text{Countermeasure} \longrightarrow \text{Countermeasure} \]

→ Countermeasures are manually superposed

→ Interactions between countermeasures are not considered

And yet

[Regazzoni et al. 2008] and [Luo et al. 2014] have demonstrated that a code protected against fault attacks may become more vulnerable to side channel attacks.
1 Composition approach
Instead of 3 Attacks
We propose 3 Attacks Countermeasures

2 Compilation approach
Source code Compiler Binary code

Thierno Barry European LLVM Developers Meeting 2017 Saarbrücken, Germany
Fault Injection Attacks
• A fault may occur at different levels

<table>
<thead>
<tr>
<th>FAULT LEVEL</th>
<th>FAULT MODEL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Algorithmic</td>
<td>Replace an instruction</td>
</tr>
<tr>
<td>Instruction</td>
<td>Replace an instruction</td>
</tr>
<tr>
<td>Register</td>
<td>Replace an instruction</td>
</tr>
<tr>
<td>Transistor</td>
<td>Replace an instruction</td>
</tr>
</tbody>
</table>

• If replaced by NOP or equivalent

<table>
<thead>
<tr>
<th>OBSERVED EFFECT</th>
<th>COUNTERMEASURE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction skip</td>
<td>Redundancy</td>
</tr>
</tbody>
</table>

• If replaced by JUMP or equivalent

<table>
<thead>
<tr>
<th>OBSERVED EFFECT</th>
<th>COUNTERMEASURE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Control flow hijacking</td>
<td>Control flow Integrity</td>
</tr>
</tbody>
</table>

• Our implemented countermeasure resists against:
  • Multi-fault that lead to skip N instructions
  • Fault that leads to skip W bytes
  • Control flow hijacking

N and W are arguments of our compiler
Instruction redundancy
TOLERANCE SCHEME

“An instruction is idempotent when it can be re-executed several times with always the same result”

**Instructions (Instr)**

\[ \forall I \in \text{Instr} \]

- **Idempotent?**
  - Yes: Duplicate \( I \)
  - No: Transform \( I \)

**EXAMPLE**

- **Idempotent**
  - `add R0, R1, R2`
  - Transformation: [Moro et al. 2014]
  - Duplication:
    - `add R0, R1, R2`
    - `add R0, R1, R2`

- **Not Idempotent**
  - `add R1, R1, R2`
  - Transformation: [Moro et al. 2014]
  - **X**
    - `add R1, R1, R2`
    - `add R1, R1, R2`

**LIMITATIONS**

- **How to find free registers at this level**
  - For [Barenghi et al. 2010]
    - The number of free registers are known for their implemented AES
  - For [Moro et al. 2014]
    - Using the ARM scratch register \( r12 \)

- **Overhead**
  - At least \( \times 4 \) for each non-idempotent instruction
  - [Moro et al. 2014] reported \( \times 14 \) for `umla1`
The internal structure of our compiler is

- Source Code
- Front-end
- IR Optimizer's
- IR
- Control flow Integrity
- Instruction Selection
- Register Allocation
- Transformation passes
- Instruction Redundancy
- Instruction Scheduling
- Instruction Separation
- Code Emission
- Binary Code

- Modified passes
- Implemented passes
The internal structure of our compiler is:

- **Source Code**
- **Front-end**
- **IR Optimizers**
- **Control flow Integrity**
- **Instruction Selection**
- **Register Allocation**
- **Transformation passes**
- **Instruction Redundancy**
- **Instruction Scheduling**
- **Instruction Separation**
- **Code Emission**
- **Binary Code**

This pass is modified in such a way that idempotent instructions are the ones privileged during the selection.

**EXAMPLE**

For the operation: \( a \times b + c \)

- \( \text{mul} \) and \( \text{add} \) are selected instead of \( \text{mla} \)

\( \text{mla} \) is not idempotent

But \( \text{mul} \) and \( \text{add} \) can be idempotent if the source and destination registers are different.
The internal structure of our compiler is

This pass is modified to introduce a constraint so that: destinations registers are always different to sources ones

EXAMPLE

For the operation: \( a = b + c \)

Instead of having: \text{add R0, R0, R1}

We have something like: \text{add R0, R1, R2} \\
\begin{align*}
\text{Duplication} & \quad \text{add R0, R1, R2} \\
\text{add R0, R1, R2} & \quad \text{add R0, R1, R2}
\end{align*}
The internal structure of our compiler is

- Source Code
- Front-end
- IR Optimizers
- IR
- Control flow Integrity
- Instruction Selection
- Register Allocation
- Transformation passes
- Instruction Redundancy
- Instruction Scheduling
- Instruction Separation
- Code Emission
- Binary Code

**BL Elimination Pass**

The role of these passes is to handle instructions that need special treatments.

- `bl fun` (Branch Link)
- `add R0, R1, R2`

**Example:**
- `bl fun`
- `add R0, R1, R2`
- `adr RX, retBB`
- `add LR, RX, #1`
- `b fun`
- `b fun`
- `retBB: add R0, R1, R2`
• The internal structure of our compiler is

![Diagram of the internal structure of the compiler]

**Example:**

\[
\begin{align*}
\text{add} & \quad R0, R1, R2 \\
\text{add} & \quad R0, R1, R2 \\
\text{ldr} & \quad R3, [R1, \#4] \\
\text{ldr} & \quad R3, [R1, \#4]
\end{align*}
\]

**Advantages:**

1. Performance
2. Security

→ to prevent faulting the original and duplicated instruction simultaneously
The internal structure of our compiler is

The role of this pass is to leave the required distance between redundant instructions to protect against fault models for which the with > size of an instruction.

**EXAMPLE**

- [Moro et al. 2014]: protects against fault that are \( \geq 32 \)-bit of width on an ARM Cortex-M3 \( \rightarrow \) 16-bit instructions are disabled \( \Rightarrow \) ++ code size

- [Rivière et al. 2015]: successfully injected faults that are \( = 64 \)-bit of width \( \rightarrow \) Moro et al’s solution doesn’t work

**Our scheme resists against both of these attack models**
- Without disabling 16-bit instructions encoding
- By simply providing the right parameters to our compiler
EXPERIMENTAL EVALUATION

- Comparison with Moro et al.’s result, using the same benchmarks and same architecture
- Target architecture: ARM Cortex-M3  
  Benchmark: AES (MiBench)  
  Size: bytes

**Performance Evaluation**

<table>
<thead>
<tr>
<th>Opt. flags</th>
<th>Overhead</th>
<th>Execution time</th>
<th>size</th>
</tr>
</thead>
<tbody>
<tr>
<td>O0</td>
<td>× 1.66</td>
<td>× 2.28</td>
<td></td>
</tr>
<tr>
<td>O3</td>
<td>× 1.98</td>
<td>× 2.16</td>
<td></td>
</tr>
</tbody>
</table>

- **Moro et al 2014**
  - Execution time: × 2.14
  - Size: × 3.02

**Security Evaluation**

- We successfully resisted against the following models of fault injections
  - Single fault that skips one instruction
  - Single fault that skips one W-instruction
  - N simultaneous faults where each fault skips one instructions
  - N simultaneous faults where each fault skips W-instructions
  - Control flow hijacking

**COMPAARED TO Moro et al.**

- **Best case**: we are 22% better in execution speed and 25% in code size
- **Worst case**: 6% better in execution speed and 26% better in code size
Thanks for your attention

Thierno Barry
PhD Candidate in Security of Embedded Systems at CEA
thierno.barry@cea.fr
http://thiernobarry.fr