AUTOMATED SOFTWARE PROTECTION FOR THE MASSES AGAINST SIDE-CHANNEL ATTACKS

Nicolas Belleville ¹
Damien Couroussé ¹
Karine Heydemann ²
Henri-Pierre Charles ¹

¹ Univ Grenoble Alpes, CEA, List, F-38000 Grenoble, France
firstname.lastname@cea.fr

² Sorbonne Université, CNRS, LIP6, F-75005, Paris, France
firstname.lastname@lip6.fr
SIDE CHANNEL ATTACKS

Electromagnetic emissions

Power consumption ...

Ciphertexts produced

Try to find the key using:
- measurements
- ciphertexts or plaintexts
- a consumption model
SIDE CHANNEL ATTACKS

Try to find the key using:
- measurements
- ciphertexts or plaintexts
- a consumption model

Electromagnetic emissions
Power consumption
Ciphertexts produced

???
SIDE CHANNEL ATTACKS

Electromagnetic emissions

Power consumption ...

Ciphertexts produced

Try to find the key using:
- measurements
- ciphertexts or plaintexts
- a consumption model

x10
SIDE CHANNEL ATTACKS

Electromagnetic emissions
Power consumption ...

Ciphertexts produced

Try to find the key using:
• measurements
• ciphertexts or plaintexts
• a consumption model
SIDE CHANNEL ATTACKS

Electromagnetic emissions

Power consumption ...

Ciphertexts produced <5kB

(less than a hello world binary file)

Try to find the key using:
• measurements
• ciphertexts or plaintexts
• a consumption model

x290
SIDE CHANNEL ATTACKS

Electromagnetic emissions

Power consumption...

Ciphertexts produced <5kB

(less than a hello world binary file)

Try to find the key using:

- measurements
- ciphertexts or plaintexts
- a consumption model

Make this bigger!
SOFTWARE COUNTERMEASURES

**Hiding**
- Secret: 01001101
- Add noise

**Masking**
- Secret: 01001101
- Split secret into shares
  - Share 1: 10010001
  - Share 2: 11011100
- Share 1 xor Share 2 = Secret

Measurements are noisy

Attack is harder

Measurements are no more directly correlated to secret
SOFTWARE COUNTERMEASURES

**Hiding**

- secret: 01001101
- add noise

**Masking**

- split secret into shares
- share 1: 10010001
- share 2: 11011100
- share1 xor share2 = secret

Measurements are noisy

Attack is harder

Measurements are no more directly correlated to secret

Attack is harder
Electromagnetic emissions
Power consumption

Function’s result

don’t change this
CODE POLYMORPHISM
WITH RUNTIME CODE GENERATION

Electromagnetic emissions
Power consumption

depends on

Executed instructions
Manipulated data

impacts

Function’s result

change this
don’t change this
CODE POLYMORPHISM
WITH RUNTIME CODE GENERATION

Electromagnetic emissions
Power consumption

impacts

Executed instructions
Manipulated data

impacts

Function’s result

don’t change this

change this
Electromagnetic emissions
Power consumption

impacts

Executed instructions
Manipulated data

impacts

Function's result

change this

change this

don’t change this
CODE POLYMORPHISM
WITH RUNTIME CODE GENERATION

Electromagnetic emissions
Power consumption
change this

Executed instructions
Manipulated data
change this

Function’s result
don’t change this
CODE POLYMORPHISM
WITH RUNTIME CODE GENERATION

Electromagnetic emissions
Power consumption
change this

Executed instructions
Manipulated data
change this

Function’s result
don’t change this

Runtime code generation
regenerate a different code regularly
only use code transformations that preserve program semantics
# CODE POLYMORPHISM WITH RUNTIME CODE GENERATION

## Issues

<table>
<thead>
<tr>
<th>Issues</th>
<th>Our contributions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Countermeasures are usually <strong>manually</strong> applied</td>
<td><strong>Automatic</strong> application of the countermeasure</td>
</tr>
<tr>
<td>Countermeasures are usually given for <strong>particular</strong> ciphers</td>
<td><strong>Any</strong> code can be hardened</td>
</tr>
</tbody>
</table>
| Target a **wide** range of platforms (be lightweight)                | Use **static** memory allocation
|                                                                      | Allocation of a **realistic** size (don’t waste memory)                           |
|                                                                      | Use **specialized** code generation                                               |
| An attacker may **write** on an **executable** memory section        | Use the **specialization** of the generator to manage memory permissions           |
| Hard to have a **good trade-off** between security and performance   | Highly **configurable** $\rightarrow$ possible to find a trade-off                |
SIDE CHANNEL ATTACKS

Unprotected

Electromagnetic emissions

Power consumption ...

Ciphertexts produced
5kB

(less than a hello world binary file)

Try to find the key using:
- measurements
- ciphertexts or plaintexts
- a consumption model

Unprotected

Electromagnetic emissions

Power consumption ...

Ciphertexts

(5kB)

(less than a hello world binary file)

Try to find the key using:
- measurements
- ciphertexts or plaintexts
- a consumption model

Unprotected

Electromagnetic emissions

Power consumption ...

Ciphertexts

(5kB)

(less than a hello world binary file)

Try to find the key using:
- measurements
- ciphertexts or plaintexts
- a consumption model
SIDE CHANNEL ATTACKS

Try to find the key using:
- measurements
- ciphertexts or plaintexts
- a consumption model

Protected

Electromagnetic emissions
Power consumption ...

Ciphertexts produced
5kB

(less than a hello world binary file)

x2.5 overhead

x290

Protected

Ciphertexts produced
5kB

(less than a hello world binary file)

x2.5 overhead

x290

Try to find the key using:
- measurements
- ciphertexts or plaintexts
- a consumption model

Protected

Electromagnetic emissions
Power consumption ...

Ciphertexts produced
5kB

(less than a hello world binary file)

x2.5 overhead

x290

Try to find the key using:
- measurements
- ciphertexts or plaintexts
- a consumption model
SIDE CHANNEL ATTACKS

Protected

Electromagnetic emissions

Ciphertexts produced
16kB

(16kB of one second of an mp3 file)

Power consumption

Try to find the key using:
- measurements
- ciphertexts or plaintexts
- a consumption model

Protected x2.5 overhead

x1 000

x1 000
SIDE CHANNEL ATTACKS

- Electromagnetic emissions
- Power consumption ...

Try to find the key using:
- measurements
- ciphertexts or plaintexts
- a consumption model

Protected

x2.5 overhead

Ciphertexts produced 160kB
(10 seconds of an mp3 file)

x10 000

Protected???
SIDE CHANNEL ATTACKS

Try to find the key using:
- measurements
- ciphertexts or plaintexts
- a consumption model

Protects
x2.5 overhead

Electromagnetic emissions
Power consumption...

Ciphertexts produced 1.6MB
(our paper)

x100 000
SIDE CHANNEL ATTACKS

- Electromagnetic emissions
- Power consumption...

Protected
x2.5 overhead

Ciphertexts produced 16MB
(a 20Mpixels jpeg image)

Try to find the key using:
- measurements
- ciphertexts or plaintexts
- a consumption model

???
SIDE CHANNEL ATTACKS

Protected
x2.5 overhead
Electromagnetic emissions
Power consumption ...
Ciphertexts produced 60MB
(a 3 minutes long full HD video)

Try to find the key using:
• measurements
• ciphertexts or plaintexts
• a consumption model

 Protected

x2.5 overhead

Electromagnetic emissions

Power consumption ...

Ciphertexts produced 60MB
(a 3 minutes long full HD video)

Try to find the key using:
• measurements
• ciphertexts or plaintexts
• a consumption model

Protected

Electromagnetic emissions

Power consumption ...

Ciphertexts produced 60MB
(a 3 minutes long full HD video)

Try to find the key using:
• measurements
• ciphertexts or plaintexts
• a consumption model
SIDE CHANNEL ATTACKS

Try to find the key using:

• measurements
• ciphertexts or plaintexts
• a consumption model

Protected

Electromagnetic emissions
Power consumption

Ciphertexts produced 60MB
(x2.5 overhead)

(x3 800 000)

Protected

x13 000 improvement!

(a 3 minutes long full HD video)
• **Background**
  - Side channel attacks
  - Software countermeasures
  - Code polymorphism

• **Automated application of code polymorphism**
  - Overview
  - Code transformations used
  - Memory management

• **Experimental evaluation**
  - Performance evaluation
  - Security evaluation
• Background
  • Side channel attacks
  • Software countermeasures
  • Code polymorphism

• Automated application of code polymorphism
  • Overview
  • Code transformations used
  • Memory management

• Experimental evaluation
  • Performance evaluation
  • Security evaluation
Main idea:
The annotated function is replaced by a wrapper and a generator.
Main idea:
Each annotated function has its own generator (with shared code segments)
Main idea:
Each annotated function has its own generator (with shared code segments)
OVERVIEW

STATICALLY

Main idea:
Each annotated function has its own generator (with shared code segments)
OVERVIEW

STATICALLY

Annotated function → Modified compiler → .c → .c → Compiler → binary

Wrapper
Specialized runtime code generator
OVERVIEW

STATICALLY

Annotated function

Modified compiler

Wrapper

Specialized runtime code generator

Compiler

binary

RUNTIME

Main idea:
At runtime, a new polymorphic instance is generated at each call

Wrapper

Runtime code generator

① calls

② generates

polymorphic instance

③ calls

polymorphic instance

polymorphic instance

polymorphic instance

polymorphic instance
How to find a good trade-off between security and performance?

How to have variability in between generations?

Main idea:
At runtime, a new polymorphic instance is generated at each call once in a while
Main idea:
The size of polymorphic instances vary

How to allocate memory?
All standard compiler optimizations are applied → the runtime generated code is statically optimized

We emit C code instead of ARM Thumb assembly
We perform static analysis to help the runtime code generator (memory and register)
#pragma odo_polymorphic
int f_critical(int a, int b) {
    int c = a^b;
    a = a+b;
    return code_f(a, b);
}

void SGPC_f_critical() {
    raise_interrupt_rm_X_add_W(code_f);
    reg_t r[] = {0,1,2,3,4,5,6,...,12,13,14,15};
    push_T2callee_saved_registers();
    eor_T2(r[4], r[1], r[0]);
    add_T2(r[0], r[1], r[0]);
    sdiv_T2(r[1], r[0], r[4]);
    mls_T2(r[0], r[1], r[4], r[0]);
    pop_T2callee_saved_registers();
    raise_interrupt_rm_W_add_X(code_f);
}
• **Background**
  • Side channel attacks
  • Software countermeasures
  • Code polymorphism

• **Automated application of code polymorphism**
  • Overview
  • Code transformations used
  • Memory management

• **Experimental evaluation**
  • Performance evaluation
  • Security evaluation
<table>
<thead>
<tr>
<th>CODE TRANSFORMATIONS USED AT RUNTIME</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Register shuffling</strong></td>
</tr>
<tr>
<td>RANDOM general purpose 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 in a RANDOM order</td>
</tr>
<tr>
<td>- xor r6, r5, r8</td>
</tr>
<tr>
<td>- add r4, r4, r5</td>
</tr>
</tbody>
</table>

**Semantic variants**
replacement of an instruction by a RANDOMLY selected semantic variant
- add r4, r4, r5
- xor r6, r5, #12348
- xor r6, r6, r8
- xor r6, r6, #12348

**Noise instructions**
insertion of a RANDOM number of RANDOMLY chosen noise instructions
- add r4, r4, r5
- sub r7, r6, r2
- load r3, r10, #53
- xor r6, r5, r8

**Dynamic noise**
RANDOM insertion of noise instructions with a RANDOM jump
- add r4, r4, r5
- jump 0, 1 or 2 instructions
- sub r7, r6, r2
- load r3, r10, #53
- xor r6, r5, r8

useless instructions
### Configurability

<table>
<thead>
<tr>
<th>Period of regeneration ( \mathbb{N} )</th>
<th>Register shuffling ( {0, 1} )</th>
<th>Instruction shuffling ( {0, 1} )</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Semantic variants</strong> ( {0, 1, 2} )</td>
<td><strong>Noise instructions</strong> ( {0, 1, 2} \times \mathbb{R} \times \mathbb{N} )</td>
<td><strong>Dynamic noise</strong> ( \mathbb{N} )</td>
</tr>
</tbody>
</table>

Total configuration space:
\[ \{0, 1\}^2 \times \{0, 1, 2\}^2 \times \mathbb{R} \times \mathbb{N}^3 \]
• Background
  • Side channel attacks
  • Software countermeasures
  • Code polymorphism

• Automated application of code polymorphism
  • Overview
  • Code transformations used
  • Memory management
    • Memory allocation & overflow prevention
    • Memory permissions

• Experimental evaluation
  • Performance evaluation
  • Security evaluation
OVERVIEW

STATICALLY

.\texttt{c} \rightarrow \textit{Modified compiler} \rightarrow \textit{.c} \rightarrow \textit{Compiler} \rightarrow \textit{binary}

\begin{itemize}
  \item \texttt{Annotated function}
  \item \texttt{Wrapper}
  \item \texttt{Specialized runtime code generator}
\end{itemize}

RUNTIME

\begin{itemize}
  \item \texttt{Wrapper}
  \item \texttt{Runtime code generator}
  \item \texttt{polymorphic instance}
\end{itemize}

Main idea:
The size of polymorphic instances vary

How to allocate memory?
MEMORY STATIC ALLOCATION

Distribution of generated codes’ size
(type of curve one can obtain)

Worst case is terrible

Probability

Amount of used memory
MEMORY STATIC ALLOCATION

Probability

Amount of used memory

Distribution of generated codes’ size
(type of curve one can obtain)

Worst case is terrible

Amount to allocate

Range where an overflow is possible

Compute a more realistic size using a threshold

threshold: probability of having an overflow $10^{-6}$ by default

For a 100 instructions code, allocated size is $5x$ smaller than worst case!
(configuration low (defined later))
OVERFLOW PREVENTION

STATICALLY

1. Modified compiler
2. Compiler
3. Binary

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)

RUNTIME

1. Wrapper
2. Runtime code generator
3. Polymorphic instance

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)
• **Background**
  - Side channel attacks
  - Software countermeasures
  - Code polymorphism

• **Automated application of code polymorphism**
  - Overview
  - Code transformations used
  - Memory management
    - Memory allocation & overflow prevention
    - Memory permissions

• **Experimental evaluation**
  - Performance evaluation
  - Security evaluation
Objective: Guarantee $W \oplus X$ and that only the generator can write into the buffer.
Background
- Side channel attacks
- Software countermeasures
- Code polymorphism

Automated application of code polymorphism
- Overview
- Code transformations used
- Memory management

Experimental evaluation
- Performance evaluation
- Security evaluation
15 different test cases
4 different selected configurations
  - none: no polymorphism
  - low: only noise instructions, generation is done every 250 executions
    - Theoretical number of variants is already very high!
      >6×10^{22} variants for a 10 instructions code
      >10^{704} variants for the 278 instructions AES we use
  - medium: all transformations activated, generation is done every execution
  - high: all transformations activated, different probability model for noise instructions insertion, generation is done every execution
STM32 board (ARM cortex M3 – 24 MHz – 8kB of RAM)
## PERFORMANCE EVALUATION

### Configuration vs. Overheads

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Execution time overhead (geometric mean)</th>
<th>Size overhead (geometric 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 generation done in linear complexity.

More results in our paper.

---

![Graph](image-url)

- \( f'(x) = 709 \) cycles per instruction
- \( f'(x) = 432 \) cycles per instruction
- \( f'(x) = 209 \) cycles per instruction
- \( f'(x) = 22 \) cycles per instruction
OUTLINE

• Background
  • Side channel attacks
  • Software countermeasures
  • Code polymorphism

• Automated application of code polymorphism
  • Overview
  • Code transformations used
  • Memory management

• Experimental evaluation
  • Performance evaluation
  • Security evaluation
- CPA on Sbox output with HW
- Success rate at 0.8 in
  - 290 traces for unprotected AES
  - 3,800,000 traces for configuration low

**Security Evaluation**

Technical details:
PicoScope 2208A, EM probe RF-U 5-2 (Langer), PA 303 preamplifier (Langer)
Sampling at 500 Msample/s with 8 bits resolution, 24,500 samples per trace

13000x more traces needed!

Execution time overhead: x2.5 including generation cost!

More results in our paper
• Automatic

• Configurable

• Efficient

• With static memory allocation of a realistic size

• With memory permission management

• Usable on constrained devices

• Open question: interest of code polymorphism against fault injection attacks?
Automated software protection for the masses against side-channel attacks

Nicolas Belleville
Damien Couroussé
Karine Heydemann
Henri-Pierre Charles

Thank you for your attention

Questions?

contact:
nicolas.belleville@cea.fr

This work was partially funded by the French National Research Agency (ANR) as part of the projects COGITO and PROSECCO, respectively funded by the programs INS-2013 under grant agreement ANR-13-INSE-0006-01 and AAP-2015 under grant agreement ANR-15-CE39.