THE MULTIPLE WAYS TO AUTOMATE THE APPLICATION OF SOFTWARE COUNTERMEASURES AGAINST PHYSICAL ATTACKS: PITFALLS AND GUIDELINES

Belleville Nicolas ¹
Barry Thierno ¹
Serai Abderrahmane ¹
Couroussé Damien ¹
Heydemann Karine ²
Robisson Bruno ³
Charles Henri-Pierre ¹

¹ Univ Grenoble Alpes, CEA, List, F-38000 Grenoble, France
firstname.lastname@cea.fr
² Sorbonne Universités, UPMC, Univ. Paris 06, CNRS,LIP6,UMR 7606 75005 Paris, France
firstname.lastname@lip6.fr
³ CEA/EMSE, Secure Architectures and Systems Laboratory CMP, 880 Route de Mimet, 13541 Gardanne, France
firstname.lastname@cea.fr
INTRODUCTION

• In 2008, for an average person: 230 embedded chips used every day!

• Number of Cyber-Physical Systems is expected to grow

• Lots of them…
  • Connected watches
  • Connected buildings
  • Smartphones
  • Monitors for human health in hospitals
  • …

• … manipulate sensitive data
  • Where you are
  • Messages between you and someone else
  • Pictures / videos of you or your house
  • Health data
  • …
• Encryption is used to protect this data
  • Secure transfers of data between connected objects and servers or cloud
  • Once encrypted, data cannot be recovered without the key

• Cryptanalysis: The designs of encryption algorithms used are well studied
  • Security relatively to attacker’s means
  • Lot of research teams try to break them
  • Their designs are a lot studied!
INTRODUCTION: CRYPTOGRAPHY

- **Black box assumption**
  - the attacker has no physical access to the key, nor to any internal processing, but can only observe external information and behavior
INTRODUCTION: CRYPTOGRAPHY

- **Black box assumption**
  - the attacker has no physical access to the key, nor to any internal processing, but can only observe *external information and behavior*
PHYSICAL ATTACKS: SIDE-CHANNEL ATTACKS

• In reality: grey box
  • Side channel information leakage:

![Side channels diagram]

Plaintext → Ciphertext

Grey box

Key

Side channels:
- Power consumption
- Electromagnetic emission
- Acoustic emission
- Time of execution
...
• In reality: grey box
  • Side channel information leakage
  • System vulnerable to faults

**Fault injection:**
- Clock glitch
- Laser beam
- Light beam
- Heating

**Plaintext**

**Bad ciphertext**

**Grey box**
• Encryption is used to protect this data
  • Secure transfers of data between connected objects and servers or cloud
  • Once encrypted, data cannot be recovered without the key

• Cryptanalysis: The designs of encryption algorithms used are well studied
  • Security relatively to attacker’s means
  • Lot of research teams try to break them
  • Their designs are a lot studied!

• Physical attacks are the only effective way to break cryptanalysis-resistant crypto ciphers
  • That’s why their countermeasures are usually evaluated on crypto blocks
  • But their range of target is BROADER than that
• Introduction

• Side channel attacks detailed example:
  how correlation power analysis works
• Fault injection attacks detailed example:
  how differential fault attacks works

• Hardware countermeasures
• Software countermeasures
  Why we want to apply them automatically
  Survey of existing approaches to apply some of them automatically
  Why we should take the compiler into account while applying countermeasure
  Why applying countermeasures within compilation process is valuable

• Conclusion
PHYSICAL ATTACKS: SIDE-CHANNEL ATTACKS

Grey box

Plaintext

Ciphertext

Key

Side channels:
- Power consumption
- Electromagnetic emission
- Acoustic emission
- Time of execution
...

 Plaintext

Key

Ciphertext
• **General approach:**
  - Divide and conquer: the key is recovered bit by bit or byte by byte
  - The attacker has a model of the electrical consumption / electromagnetic emission /…

• **Attack steps:**
  - Choose a target intermediate value
    - That depends only of one byte of the key ideally
**General approach:**
- Divide and conquer: the key is recovered bit by bit or byte by byte
- The attacker has a model of the electrical consumption / electromagnetic emission /…

**Attack steps:**
- Choose a target intermediate value
- Compute a theoretical emission for this value for all key hypothesis
  - With a model of emission (hamming weight or hamming distance usually used)
  - The theoretical emission is computed for all key hypothesis for N plaintexts
  - We get Nx256 theoretical emissions (attack of one byte of the key)
**General approach:**
- Divide and conquer: the key is recovered bit by bit or byte by byte
- The attacker has a model of the electrical consumption / electromagnetic emission /…

**Attack steps:**
- Choose a target intermediate value
- Compute a theoretical emission for this value for all key hypothesis
  - With a model of emission (hamming weight or hamming distance usually used)
  - The theoretical emission is computed for all key hypothesis for N plaintexts
  - We get Nx256 theoretical emissions (attack of one byte of the key)
• **General approach:**
  • Divide and conquer: the key is recovered bit by bit or byte by byte
  • The attacker has a model of the electrical consumption / electromagnetic emission /…

• **Attack steps:**
  • Choose a target intermediate value
  • Compute a theoretical emission for this value for all key hypothesis
  • Measure emission through several encryptions
    • At least one encryption per plaintext
    • Measurements have to be aligned
General approach:
- Divide and conquer: the key is recovered bit by bit or byte by byte
- The attacker has a model of the electrical consumption / electromagnetic emission /…

Attack steps:
- Choose a target intermediate value
- Compute a theoretical emission for this value for all key hypothesis
- Measure emission through several encryptions
- Compare measurements with theoretical values
  - Highest correlation between theory and traces gives a key candidate
• General approach:
  - Divide and conquer: the key is recovered bit by bit or byte by byte
  - The attacker has a model of the electrical consumption / electromagnetic emission / …

• Attack steps:
  - Choose a target intermediate value
  - Compute a theoretical emission for this value for all key hypothesis
  - Measure emission through several encryption
  - Compare measurements with theoretical values
  - Highest correlation between theory and traces gives a key candidate

This is an example of how side channel attacks can be mounted.

BUT: they can target other kind of applications (web browsers, verifypin, …), and can also be used to help monitoring fault injection attacks
PHYSICAL ATTACKS: FAULT INJECTION ATTACKS

Fault injection:
- Clock glitch
- Laser beam
- Light beam
- Heating
• **General approach:**
  - Divide and conquer: the key is recovered bit by bit or byte by byte
  - Perform a fault during encryption
  - The encryption will generate a bad ciphertext
  - Compare the bad ciphertext with the reference one
• **Attack steps:**
  - Choose a target instruction or data
**General approach:**
- Divide and conquer: the key is recovered bit by bit or byte by byte
- Perform a fault during encryption
- The encryption will generate a bad ciphertext
- Compare the bad ciphertext with the reference one

**Attack steps:**
- Choose a target instruction or data
- Compute the effect of the fault for all keys and plaintexts on the ciphertext
  - Use a model of the fault like instruction skip or data nullified

---

**Diagram:**
- AES
- Plaintexts
- Keys
- Fault to be injected here
- Nx256
- Bad ciphertexts
**General approach:**
- Divide and conquer: the key is recovered bit by bit or byte by byte
- Perform a fault during encryption
- The encryption will generate a bad ciphertext
- Compare the bad ciphertext with the reference one

**Attack steps:**
- Choose a target instruction or data
- Compute the effect of the fault for all keys and plaintexts on the ciphertext
- Collect the ciphertexts for all plaintexts while faulting the chip
General approach:
- Divide and conquer: the key is recovered bit by bit or byte by byte
- Perform a fault during encryption
- The encryption will generate a bad ciphertext
- Compare the bad ciphertext with the reference one

Attack steps:
- Choose a target instruction or data
- Compute the effect of the fault for all keys and plaintexts on the ciphertext
- Collect the ciphertexts for all plaintexts while faulting the chip
- Compare ciphertexts obtained with the theoretical ones
PHYSICAL ATTACKS: FAULT INJECTION ATTACKS

- General approach:
  - Divide and conquer: the key is recovered bit by bit or byte by byte
  - Perform a fault during encryption
  - The encryption will generate a bad ciphertext
  - Compare the bad ciphertext with the reference one

- Attack steps:
  - Choose a target instruction or data
  - Compute the effect of the fault for all keys and plaintexts on the ciphertext
  - Collect the ciphertexts for all plaintexts while faulting the chip
  - Compare ciphertexts obtained with the theoretical ones

This is an example of how fault injections can be used.

BUT: they can target other kind of applications! (bootloaders, verifypin, …)
COUNTERMEASURES

• **Side-channel:**
  • **Hiding**
    • Lower the SNR (Signal Noise Ratio) in measurements
  • **Masking**
    • Break the direct link between emissions and the key

• **Fault injection attacks:**
  • **Fault tolerance**
    • A fault won’t change the behavior of the program
  • **Fault detection**
    • A fault will be detected and put the program/chip in a predefined state

The key is qshgoq

The key combined with randomness is qshgoq

I’m fine

Good output

Sorry, I’m on sick leave

No output
HARDWARE COUNTERMEASURES

- **Side-channel:**
  - Dual rail with precharge logic
    - 0 and 1 are encoded with (0,1) and (1,0) couples
    - Output of each gate is precharged with either (0,0) or (1,1)
    - Hamming weight and Hamming distance are independent of data

- Insert noise
  - Random voltage scaling
  - Variable clock speed (temporal desynchronization)

- Filter power consumption
  - Make the power consumption as constant as possible

\[
\begin{align*}
\text{HW}(01) &= 1 \\
\text{HW}(10) &= 1 \\
\text{HD}(00,01) &= 1 \\
\text{HD}(00,10) &= 1
\end{align*}
\]
HARDWARE COUNTERMEASURES

- **Fault injection attacks:**
  - Encapsulation
    - Prevent the attack by making the access to components hard

- Detector of light emission / magnetic field
  - Detect signals which may be related to a fault injection

- Integrity
  - Check the absence of control flow corruption (CFI)
  - Check data integrity

- Error correcting memory
  - The memory is able to correct a certain number of errors in the data
HARDWARE COUNTERMEASURES

• **Side-channel:**
  - Dual rail with precharge logic
  - Insert noise
  - Filter power consumption

• **Fault injection attacks:**
  - Encapsulation
  - Detector of light emission / magnetic field
  - Control flow integrity
  - Error correcting memory

• **Problems / Limitations:**
  - Requires expertise
  - Takes time to implement
  - Costly hardware
  - Impossible to update
  - Countermeasure is applied everywhere, even on uncritical code
SOFTWARE COUNTERMEASURES

• Side-channel:
  • Instructions shuffling & Temporal desynchronization
    • Make alignment of measurements fail
    • Dependency analysis between instructions based on registers used or defined
    
    ```
    for (i=0; i<n; i++) {
      k = rand(possible_values);
      T[k]=T[k]+1;
    }
    iterate in random order
    ```
  
  • Masking
    • Combine the key with a random number to change the profile of the leakage
    • All the algorithm is modified so that everything is computed using the masked key
    
    ```
    mask = rand();
    masked_key = key xor mask;
    a = a xor key;
    b = a;
    return b;
    ```
    choose randomly at runtime
    between the 2 forms
    
    ```
    if (rand(2)) {
        add r3, r3, #1
        sub r6, r7, #3
    } else {
        sub r6, r7, #3
        add r3, r3, #1
    }
    choose randomly at runtime
    between the 2 forms
    ```

  everything is computed masked
  the mask is removed from the result at the end
SOFTWARE COUNTERMEASURES

• Fault injection attacks:
  • Code duplication
    • Some parts of the code are duplicated / Duplication of all instructions
    • Tolerance of one instruction-skip fault

```
if (password == "ok") {
  if (password == "ok") { … }
}
duplicate code
```

• Control flow integrity
  • At each basic block, check that we come from a legitimate basic block
  • Detection of instruction-modification fault that change the control flow

```
add r3, r4, #1 ↔ add r3, r4, #1
```

duplicate instructions

• Error detecting codes throughout the algorithms
  • Add a parity bit to the variables and keep trace of it
  • Detection of data-corruption fault

```
0110010101100101
Ok
Error !
```
SOFTWARE COUNTERMEASURES

• **Side-channel:**
  - Instructions shuffling & Temporal desynchronization
  - Masking

• **Fault injection attacks:**
  - Code duplication
  - Control flow integrity
  - Error detecting codes throughout the algorithms

• **Problems:**
  - Requires expertise
  - Takes time to implement
  - Implementation on every critical functions
  - Compilation can optimize out countermeasures
  - Performance cost
SOFTWARE COUNTERMEASURES

• **Side-channel:**
  - Instructions shuffling & Temporal desynchronization
  - Masking

• **Fault injection attacks:**
  - Code duplication
  - Control flow integrity
  - Error detecting codes throughout the algorithms

• **Problems:**
  - Requires expertise
  - Takes time to implement
  - Implementation on every critical functions
  - Compilation can optimize out countermeasures
  - Performance cost

Automatically apply them?
SOFTWARE COUNTERMEASURES

• **Side-channel:**
  - Instructions shuffling & Temporal desynchronization
  - Masking

• **Fault injection attacks:**
  - Code duplication
  - Control flow integrity
  - Error detecting codes throughout the algorithms

• **Problems:**
  - Requires expertise
  - Takes time to implement
  - Implementation on every critical functions
  - Compilation can optimize out countermeasures
  - Performance cost

Automatically apply them? HOW?
DIFFERENT LEVELS OF APPLICATION

Source code → Parse source code, change representation → Optimize code → Optimize code, select instructions, register allocation, emit assembly → Binary / assembly

Compilation
DIFFERENT LEVELS OF APPLICATION

- **Source code**
  - Front end: Parse source code, change representation
  - Middle end: Optimize code
  - Back end: Optimize code, select instructions, register allocation, emit assembly

Compilation

- Binary / assembly

Apply countermeasure
DIFFERENT LEVELS OF APPLICATION

Source code

Front end
Parse source code, change representation

Middle end
Optimize code

Back end
Optimize code, select instructions, register allocation, emit assembly

Binary / assembly

Compilation

Apply countermeasure
DIFFERENT LEVELS OF APPLICATION

Source code

Front end
- Parse source code, change representation

Middle end
- Optimize code

Back end
- Optimize code, select instructions, register allocation, emit assembly

Compilation

Binary / assembly

Apply countermeasure
COUNTERMEASURES: HOW TO APPLY THEM?

→ SOURCE CODE

- **Steps to do once:**
  - Write a parser
  - Write a transformation pass for critical parts
  - Write a file emitter for targeted format

- **Steps to do for every file:**
  - Transform file
  - Compile file
  - Disassemble file
  - **Check that countermeasures are still here**

- **Disabling compiler optimizations (-O0) to skip the checking phase is a bad idea**
  - Horrible performance
  - Register spilling → new leakage

- **References that use this approach:** [Eldib, LNCS, 2014] [Lalande, LNCS, 2014] [Luo, ASAP, 2015]
COUNTERMEASURES: HOW TO APPLY THEM?

→ WITHIN THE COMPILER

• **Steps to do once:**
  • Update the parser
  • Add a transformation pass to transform critical parts
  • Check once for all that later transformations do not threaten the countermeasure
  • If necessary, deactivate or transform some of them

• **Steps to do for every file:**
  • Compile file

  no need to be a security expert here

• **The code resulting is correctly optimized**

• **References that use this approach:** [Agosta, IEEE TCAD, 2015] [Agosta, DAC, 2012] [Agosta, DAC, 2013] [Barry, CS2, 2016] [Bayrak, IEEE TC, 2015] [Malagón, Sensors, 2012] [Moss, LNCS, 2012]
  • [Bayrak, IEEE TC, 2015]: hybrid approach between the “assembly” and “within the compiler” approaches. Uses the compiler to **decompile** a binary file up to an intermediate representation before applying the countermeasure.
COUNTERMEASURES: HOW TO APPLY THEM?

→ ASSEMBLY CODE

• **Steps to do once:**
  - Write a parser
  - Write analysis passes which reconstruct some higher level information if necessary
  - Write the transformation
  - Write a file emitter

• **Steps to do for every file:**
  - Compile the file
  - Disassemble it
  - Transform it
  - Reassemble it

  no need to be a security expert here

• **The resulting code is secured but performance can be affected**
  - Compiler uses registers as if they won’t be used for something else
  - The need for additional registers while applying countermeasure may lead to register spilling

• **References that use this approach:** [Bayrak, DAC, 2011] [Moro, 2014] [Rauzy, JCEN, 2016]
COUNTERMEASURES: HOW TO APPLY THEM?

→ DETAILED EXAMPLES

<table>
<thead>
<tr>
<th>Level</th>
<th>Team</th>
<th>Approach</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source code</td>
<td>Lalande &amp; al.</td>
<td>• CFI applied on C code</td>
</tr>
<tr>
<td></td>
<td>Eldib &amp; al.</td>
<td>• Use clang as a parser and apply Masking with a SMT solver</td>
</tr>
<tr>
<td>Within the compiler</td>
<td>Agosta &amp; al.</td>
<td>• Modified LLVM (new passes &amp; modified passes). Hiding applied automatically.</td>
</tr>
<tr>
<td></td>
<td>Barry &amp; al.</td>
<td>• Modified LLVM (new passes &amp; modified passes). Instruction Duplication applied automatically.</td>
</tr>
<tr>
<td>Assembly code</td>
<td>Bayrak &amp; al.</td>
<td>• Random precharging applied automatically.</td>
</tr>
<tr>
<td></td>
<td>Moro &amp; al.</td>
<td>• Instruction Duplication applied automatically.</td>
</tr>
</tbody>
</table>

## COUNTERMEASURES: HOW TO APPLY THEM?

→ DETAILED EXAMPLES

<table>
<thead>
<tr>
<th>Level</th>
<th>Team</th>
<th>Approach</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source code</td>
<td>Lalande &amp; al.</td>
<td>• CFI applied on C code</td>
</tr>
<tr>
<td></td>
<td>Eldib &amp; al.</td>
<td>• Use clang as a parser and apply Masking with a SMT solver</td>
</tr>
<tr>
<td>Within the compiler</td>
<td>Agosta &amp; al.</td>
<td>• Modified LLVM (new passes &amp; modified passes). Hiding applied automatically.</td>
</tr>
<tr>
<td>Barry &amp; al.</td>
<td></td>
<td>• Modified LLVM (new passes &amp; modified passes). Instruction Duplication applied automatically.</td>
</tr>
<tr>
<td>Assembly code</td>
<td>Bayrak &amp; al.</td>
<td>• Random precharging applied automatically.</td>
</tr>
<tr>
<td>Moro &amp; al.</td>
<td></td>
<td>• Instruction Duplication applied automatically.</td>
</tr>
</tbody>
</table>

For the same countermeasure, compiler approach reduced performance overhead from $\times 2.86$ to $\times 1.92$ and size overhead from $\times 2.90$ to $\times 1.16$ for MiBench AES
## COUNTERMEASURES: HOW TO APPLY THEM?

→ IN A NUTSHELL

<table>
<thead>
<tr>
<th>Level</th>
<th>Pros</th>
<th>Cons</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source code</td>
<td>• More or less straightforward</td>
<td>• Countermeasure can be optimized out during compilation</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Assembly code MUST be checked after compilation</td>
</tr>
<tr>
<td>Within the compiler</td>
<td>• Provide security AND performance</td>
<td>• Harder to implement.</td>
</tr>
<tr>
<td></td>
<td>• Optimizations can be controlled</td>
<td>• Requires to have access to the compiler source code</td>
</tr>
<tr>
<td>Assembly code</td>
<td>• Countermeasure not optimized out</td>
<td>• Can be hard to take all instructions into account or to do high level transformations</td>
</tr>
<tr>
<td></td>
<td>• Can even secure binary programs without their source code</td>
<td>• Performance more affected</td>
</tr>
</tbody>
</table>

CPSEd 2017 | Belleville Nicolas | 41
CONCLUSION

• Physical attacks are an important threat for cyber-physical systems
  • They are the only effective way to break encryption
  • Their range of target is broader than encryption
  • Best security levels are reached by combining hardware and software countermeasures

• Securing is costly
  • Automatic application of software countermeasures or automatic design of hardware with countermeasures can reduce this cost

• Compilation is usually forgotten in potential threats to countermeasures
  • source code ≠ binary

• Securing during compilation is valuable
  • Enables to optimize the performance cost of a countermeasure

• Hardware has to be taken into account too
  • binary ≠ what is really executed
  • Speculative execution within the processor
Physical attacks are an important threat for cyber-physical systems
  - They are the only effective way to break encryption
  - Their range of target is broader than encryption
  - Best security levels are reached by combining hardware and software countermeasures

Securing is costly
  - Automatic application of software countermeasures or automatic design of hardware with countermeasures can reduce this cost

Compilation is usually forgotten in potential threats to countermeasures
  - source code ≠ binary

Securing during compilation is valuable
  - Enables to optimize the performance cost of a countermeasure

Hardware has to be taken into account too
  - binary ≠ what is really executed
  - Speculative execution within the processor

Pay attention to these!


REFERENCES


Thank you for your attention

Questions?

THE MULTIPLE WAYS TO AUTOMATE THE APPLICATION OF SOFTWARE COUNTERMEASURES AGAINST PHYSICAL ATTACKS: PITFALLS AND GUIDELINES

Belleville Nicolas ¹
Barry Thierno ¹
Serai Abderrahmane ¹
Couroussé Damien ¹
Heydemann Karine ²
Robisson Bruno ³
Charles Henri-Pierre ¹

¹ Univ Grenoble Alpes, CEA, List, F-38000 Grenoble, France
firstname.lastname@cea.fr
² Sorbonne Universités, UPMC, Univ. Paris 06, CNRS,LIP6,UMR 7606 75005 Paris, France
firstname.lastname@lip6.fr
³ CEA/EMSE, Secure Architectures and Systems Laboratory CMP, 880 Route de Mimet, 13541 Gardanne, France
firstname.lastname@cea.fr