SECURITY OF ASSEMBLY PROGRAMS AGAINST FAULT ATTACKS ON EMBEDDED PROCESSORS

Nicolas MORO (CEA)

Thesis supervised by Karine HEYDEMANN (LIP6)
Under the direction of Emmanuelle ENCRENAZ (LIP6)
and Bruno ROBISSON (CEA)

Some parts of the presented works have been done in cooperation with Amine Dehbaoui

13TH NOVEMBER 2014 – PARIS
Embedded systems:

- Are **autonomous electronic systems**
- Are **widely used** and have many applications
• Those systems can be attacked

• Those attacks generally aim at:

Getting sensitive data

Bypassing a protection

Doing reverse-engineering
Embedded systems security is very important for:

- **Administrations and governments**
  - Digital identity documents

- **Manufacturers of smart cards**
  - Pay-TV, banking cards, access cards, …

- **Manufacturers of consumer products**
  - Locked systems, which include payment systems, …
Physical attacks
• require an **access to the component**
• aim at exploiting the **vulnerabilities of integrated circuits**
• are a **serious threat** for embedded systems
TWO KINDS OF PHYSICAL ATTACKS

• Side-channel analysis attacks

• Fault injection attacks
DIFFERENT FAULT INJECTION MEANS

- Focused light
- Electromagnetic waves
- Power supply voltage
- Temperature
- Clock signal
Some countermeasures put at **different levels**:

1 – Physical sensors

Light detectors, voltage modification detectors, ...

2 – Mécanismes de détection ou tolérance aux fautes

Redundancy

Parity bits and error correcting codes

Mathematical properties of algorithms

---

**Algorithm 3: Shamir’s countermeasure**

Input: message digest $m$, private key $p$, $q$, $d$, $i_p$.

Output: signature $S = m^{d^i_p} \mod N$.

1. begin
2. Generate a random prime $r$.
3. $S_p \leftarrow m^{d^i_p} \mod q \mod r$.
4. $S_q \leftarrow m^{d^i_p} \mod p \mod q \mod r$.
5. if $S_p \neq S_q \mod r$ then
6. Return error.
7. end
8. $S_p \leftarrow S_p \mod r$ and $S_r \leftarrow S_p \mod q$.
9. Recombine $S_n$ and $S_q$ as explained previously.
10. Return $S$.
11. end
PULSED ELECTROMAGNETIC FAULT INJECTION

- **Quite recent** technique
  (theoretical in 2002, in practice since 2007)

- An electrical pulse is sent to an **injection antenna**

- Electromagnetic coupling with the **power grid** of the circuits

- **Semi-local effect**

- **Quite easy to set-up**

Can **bypass some existing countermeasures**

OBJECTIVE OF THIS THESIS

- Some new attacks with EM injection have been achieved → **new countermeasures** are necessary

**Hardware** countermeasures
- Requires significant changes
- Only for circuit manufacturers
- Need a finished circuit for testing

**Software** countermeasures
- More flexible changes
- Can be applied to processors
- Easier to test

- Difficult to model the impact of an EM injection on the execution of a program → **assembly level**

---

**Objective of this thesis:**
Propose **software countermeasures** against electromagnetic injection attacks.
CHOSEN APPROACH FOR THIS THESIS

Definition of a **fault model**
Study of the effects of a fault injection on an assembly program

Definition of a **countermeasure**
Résistant against the faults of the model and formally verified

Experimental evaluation of its **efficiency**
Experimental tests on isolated instructions and more complex codes
CHOSEN APPROACH FOR THIS THESIS

Study and usage of the fault injection means

- Laser
  - Roscian, 2013
- EM waves
  - Dehbaoui, 2012
- Voltage glitch
  - Zussa, 2014
- Temperature
  - Schmidt, 2014
- Clock glitch
  - Balasch, 2011

Definition of fault models at a higher level

- Bit flip
  - Agoyan, 2010
- Instruction skip
  - Barenghi, 2012
- Instruction corruption
  - Balasch, 2011
  - Trichina, 2010
- Branches
  - Berthomé, 2012

Design of counter-measures based on these models

- Redundancy
  - Barenghi, 2010
  - Bar-El, 2006
- Formal methods
  - Rauzy, 2013
  - Christofi, 2013
- Java Card
  - Sere, 2011
  - Barbu, 2011
I. Introduction

II. Conception of a fault injection bench

III. Validation of a fault model at assembly level

IV. Definition and verification of a software countermeasure

V. Test and experimental evaluation of the countermeasure

VI. Conclusion and perspectives
1. The experiment is **driven from the PC**
2. The target code is **executed on the microcontroller**
3. The microcontroller **sends a trigger signal**
4. The generator **sends a voltage pulse**
5. The microcontroller is stopped
6. The internal data is harvested
• ARM architecture in the majority of embedded systems
• Several secured processors based on an ARM Cortex-M
• The Cortex-M3 architecture is already used for some smart cards or some processors for RFID communications

• Fréquence of 56 MHz, clock period 17.8 ns
• Architecture ARMv7-M 32-bit (Harvard type)
**Thumb-2 instruction set**

- RISC, 151 instructions encoded on 16 and 32 bits
- Load/store architectures: operations are performed on registers

```
add.w r1, r0, #1
```

- **Operation**: `add.w`
- **Destination register**: `r1`
- **Source register**: `r0`
- **Immediate value (constant)**: `#1`
- **Suffix to force a 32-bit encoding**
3 levels of pipeline (Fetch – Decode – Execute), no prefetch

<table>
<thead>
<tr>
<th>Fetch</th>
<th>Loading of the instruction into the instruction register</th>
</tr>
</thead>
<tbody>
<tr>
<td>Decode</td>
<td>Decoding, operand fetch, branch detection</td>
</tr>
<tr>
<td>Execute</td>
<td>Execution of the instruction, writing of the results</td>
</tr>
</tbody>
</table>

**Diagram**

- **Fetch**: Loading of the instruction into the instruction register.
- **Decode**: Decoding, operand fetch, branch detection.
- **Execute**: Execution of the instruction, writing of the results.
- The fault injection antenna is a copper coil

- **Keil ULINKpro** JTAG debug probe
  Enables to use the microcontroller in debug mode

- **Pulse** generator
  High voltage, high current

![Graph showing voltage vs. time]

**Time (ns)**

**Voltage (V)**
OUTLINE OF THE PRESENTATION

I. Introduction

II. Conception of a fault injection bench

III. Validation of a fault model at assembly level

IV. Definition and verification of a software countermeasure

V. Test and experimental evaluation of the countermeasure

VI. Conclusion and perspectives
DEFINITION OF A FAULT MODEL

• Enables to better understand the **abilities of an attacker**

• A big number of **experimental parameters**

• Their influence on the obtained faults **must be studied**

**Studied parameters**

- **Position of the antenna**
- **Injection time**
- **Pulse voltage**
INFLUENCE OF THE POSITION OF THE ANTENNA

ldr r8, =0x12345678 ➜ loads a 32-bit word from the Flash memory

• Variation of X and Y on a square with 3mm side
• Fixed voltage, fixed injection time, fixed position on the Z axis

A local effect of the fault injection technique
A very small area to inject faults
Two distinct time intervals
Different kinds of faults
For some injection times, we obtained 100% of faults

Variation of the injection time
Fixed voltage
Fixed antenna

ldr r8,=0x12345678 loads a 32-bit word from the Flash memory
INFLUENCE OF THE PULSE’S VOLTAGE

ldr r4, =0x12345678  ➔ loads a 32-bit word from the Flash memory

• Variation of the pulse voltage
• Fixed antenna, fixed injection time

<table>
<thead>
<tr>
<th>Voltage</th>
<th>Output value</th>
</tr>
</thead>
<tbody>
<tr>
<td>172V</td>
<td>1234 5678</td>
</tr>
<tr>
<td>174V</td>
<td>9234 5678</td>
</tr>
<tr>
<td>176V</td>
<td>FE34 5678</td>
</tr>
<tr>
<td>178V</td>
<td>FFF4 5678</td>
</tr>
<tr>
<td>180V</td>
<td>FFFD 5678</td>
</tr>
<tr>
<td>182V</td>
<td>FFFF 7F78</td>
</tr>
<tr>
<td>184V</td>
<td>FFFFF FFFD</td>
</tr>
<tr>
<td>186V</td>
<td>FFFFF FFFF</td>
</tr>
</tbody>
</table>

➢ A set at 1 effect on the bits
➢ The effect is related to the increase of the voltage
➢ Only obtained when loading data from the Flash memory
Transfer of an instruction on the HRDATAI instruction bus

1 – The address of the instruction is put on the HADDRI bus
Transfer of an instruction on the HRDATAI instruction bus

2a – The binary encoding of the instruction is put on the HRDATAI bus (end of the following clock cycle)
Transfer of an instruction on the HRDATAI instruction bus

2b – The address of the next instruction is put on the HADDRI bus
Transfer of an instruction on the HRDATAI instruction bus

3 – Corruption of the transfer of the instruction on the HRDATAI bus

Horloge
HADDR
HADDR
HRDATA
0x00 nop
0x02 nop
0x04 nop
0x06 nop
0x08 nop
0x0A 1dr r4,[pc,#40]
0x0C nop
0x0E nop
0x10 nop

EXECUTE
DECODE
EXECUTE
DECODE
EXECUTE
DECODE
FETCH
FETCH
FETCH
FETCH
FETCH
FETCH
FETCH

0x06
0x0A
0x0E
래시

Transfer of a piece of data on the HRDATA data bus

1 – The address of the piece of data is put on the HADDR bus
Transfer of a piece of data on the HRDATA data bus

2a – Corruption of the transfer of the piece of data on the HRDATA bus

Transfer of a piece of data on the HRDATA data bus
Transfer of a piece of data on the HRDATA data bus

2b – Corruption of the transfer of other instructions on the HRDATAI bus
The code `ldr r0, [pc,#40]` loads a 32-bit word from the Flash memory.
ldr r0, [pc, #40] ➔ loads a 32-bit word from the Flash memory

Possibility to corrupt the transfers from the Flash memory
(transfers of instructions and data)
How can we extract from the previous results a fault model for the instruction replacements?

Search for replacements that can explain the obtained faults:

• Simulation of instruction corruptions
• Comparison with the experimental results
OVERVIEW OF THE FAULT MODEL AT RTL LEVEL

Consequences regarding **instructions**

- Instruction **replacements**
- Instruction **skips** in some cases

```
| nop  | 1011 1111 0000 0000 |
| nop  | 1011 1111 0000 0000 |
| str r0, [r0, #0] | 0110 0000 0000 0000 |
| nop  | 1011 1111 0000 0000 |
```

Consequences regarding **data**

- Corruption of the ldr from the Flash memory (encryption keys, …)
- Values with high Hamming weight easier to obtain (on this target)

Electromagnetic Fault Injection: Towards a Fault Model on a 32-bit Microcontroller
N. Moro, A. Dehbaoui, K. Heydemann, B. Robisson, E. Encrenaz – FDTC 2013, Santa-Barbara, USA
« nop » fault model (instruction skip)
- Can enable to skip a subroutine call
- Possible to detect some vulnerabilities on a program

Replacement by un nop statistically more frequent
- Writing into a dead register or a unused memory address
- Re-execution of a previous idempotent instruction (add r1,r2,r3 ...)
- Replacement by an instruction without any effect (mov r0,r0 ...)

In which proportion do the injected faults have an effect that is similar to an instruction skip (nop) ?
EVALUATION THE NOP MODEL’S COVERAGE RATE

1. **Experiment** on a program that sums the elements of a 2-value array

   ```
   tab[0] = 1 ; tab[1] = 2
   ```

   **Expected result**: 3

2. **Simulation** of the skip of every instruction

3. **Comparison** between the output values and the experimental results

   **Simulation**
   ```
   boucle_addition:
   ldr r4, [r2, r1, lsl #2]
   ldr r3, [r0, #0]
   add r3, r3, r4
   str r3, [r0, #0]
   add r1, r1, #1
   cmp r1, #2
   blt boucle_addition
   ```

   ```
   r4 = array[i]
   ; r3 = result
   ; r3 = r3 + r4
   ; resultat = r3
   ; r1 = r1 + 1
   ; r1 == 2 ?
   ```
EVALUATION THE NOP MODEL’S COVERAGE RATE

1. Experiment on a program that sums the elements of a 2-value array:

```
boucle_addition:
1 ldr r4, [r2, r1, lsl #2]; r4 = array[i]
2 ldr r3, [r0, #0]; r3 = result
3 add r3, r3, r4; r3 = r3 + r4
4 str r3, [r0, #0]; resultat = r3
5 add r4, r4, r4
6 cmp r4, r4
7 blt b
```

On the tested program, about **25%** of the faults obtained in practice can be seen as an instruction skip.

2. Simulation of the skip of every instruction

3. Comparison between the output values and the experimental results:

<table>
<thead>
<tr>
<th>Interruption</th>
<th>r0</th>
<th>r1</th>
<th>r2</th>
<th>r3</th>
<th>r4</th>
<th>r5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aucune</td>
<td>0x2000040C</td>
<td>0x2</td>
<td>0x2000421</td>
<td>0x2 [FAUTE]</td>
<td>0x1 [FAUTE]</td>
<td>0x40010C10</td>
</tr>
<tr>
<td>t= 37.0 ns</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Aucune</td>
<td>0x2000040C</td>
<td>0x2</td>
<td>0x2000421</td>
<td>0x3</td>
<td>0x2</td>
<td>0x40010C10</td>
</tr>
<tr>
<td>t= 37.2 ns</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Aucune</td>
<td>0x2000040C</td>
<td>0x2</td>
<td>0x2000421</td>
<td>0x3</td>
<td>0x2</td>
<td>0x40010C10</td>
</tr>
<tr>
<td>t= 37.4 ns</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Aucune</td>
<td>0x2000040C</td>
<td>0x2</td>
<td>0x2000421</td>
<td>0x3</td>
<td>0x2</td>
<td>0x40010C10</td>
</tr>
<tr>
<td>t= 37.6 ns</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>UsageFault</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>
I. Introduction

II. Conception of a fault injection bench

III. Validation of a fault model at assembly level

IV. Definition and verification of a software countermeasure

V. Test and experimental evaluation of the countermeasure

VI. Conclusion and perspectives
- Many **instruction replacements** have an effect that is equivalent to an **instruction skip**

- Some **double faults** are possible if the time between both injections is high enough (a few µs for our bench)

How can we **guarantee a correct execution** with a potential instruction skip by an attacker?
1. Able to resist to a fault injection
   ➡ Based on a temporal redundancy principle

2. Resistant to double faults sufficiently far apart
   ➡ Instruction-level redundancy

3. Can be automatically applied
   ➡ Replacement sequence for every instruction
   ➡ Semantic equivalence regarding the initial instruction
   ➡ Principle to reinforce a full program
DIFFERENT CLASSES OF INSTRUCTIONS

**Idempotent instructions**

- `add r1, r0, #1`
- `add r1, r0, #1`

**Separable instructions**

- `add r12, r1, #1`
- `add r12, r1, #1`
- `mov r1, r12`
- `mov r1, r12`

Duplication not correct if no fault

- Separation then duplication
DIFFERENT CLASSES OF INSTRUCTIONS

Idempotent instructions

\[
\text{add} \quad r1, r0, #1
\]

Separable instructions

\[
\text{push} \quad \{r1, r2, r3, lr\}
\]

\[
\text{stmdb} \quad sp, \{r1, r2, r3, lr\}
\]

\[
\text{stmdb} \quad sp, \{r1, r2, r3, lr\}
\]

\[
\text{sub} \quad r12, sp, #16
\]

\[
\text{sub} \quad r12, sp, #16
\]

\[
\text{mov} \quad sp, r12
\]

\[
\text{mov} \quad sp, r12
\]

\[\Rightarrow\] Separation then duplication
**DIFFERENT CLASSES OF INSTRUCTIONS**

**Idempotent instructions**

- add \( r1, r0, #1 \)

**umlal**

\[ r1:r2 = r3*r4 + r1:r2 \]

*Multiplication and addition over 64 bits*

- mrs \( r12, apsr \)
- umull \( r10, r11, r3, r4 \)
- umull \( r10, r11, r3, r4 \)
- adds \( r9, r10, r1 \)
- adds \( r9, r10, r1 \)
- addc \( r10, r11, r2 \)
- addc \( r10, r11, r2 \)
- mov \( r1, r9 \)
- mov \( r1, r9 \)
- mov \( r2, r10 \)
- mov \( r2, r10 \)
- msr \( apsr, r12 \)
- msr \( apsr, r12 \)
• Branch instruction can be duplicated…
  but not subroutine call instructions
  *(otherwise every subroutine would be executed twice)*

```
adr r12, return_label
adr r12, return_label
add lr, r12, #1
add lr, r12, #1
b function
b function
```

- Puts the return address into r12
- Updates the return pointer lr
- Branches to the subfunction

---

Nicolas Moro - Thesis defense | 13th November 2014 | Page 44
Properties to verify:

1. Every replacement sequence has the same semantics than the instruction it replaces.

2. Every replacement sequence is tolerant to an instruction skip.

\[
AG[(i.pc = pc_{init_i}) \land (c.pc = pc_{init_c})] \Rightarrow AF[(i.pc = pc_{final_i}) \land (c.pc = pc_{final_c}) \land \forall x \in D, (i.x = c.x)]
\]
The verification step has a **double purpose**:

1. **Verification of the equivalence of sequences**
   (with and without fault injections)

2. **Help for the design of sequences**
   (enables to highlight the challenging cases)

\[ AG[(i.pc = pc\_init\_i) \land (c.pc = pc\_init\_c)] \Rightarrow AF[(i.pc = pc\_final\_i) \land (c.pc = pc\_final\_c) \land \forall x \in D, (i.x = c.x)] \]
• An **automatic application algorithm** has been designed

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Overhead (cycles)</th>
<th>Overhead (code size)</th>
</tr>
</thead>
<tbody>
<tr>
<td>AES</td>
<td>+ 113.7%</td>
<td>+ 202%</td>
</tr>
<tr>
<td>MiBench AES</td>
<td>+ 186.4%</td>
<td>+ 189.9%</td>
</tr>
<tr>
<td>MiBench SHA0</td>
<td>+ 122.8%</td>
<td>+ 178.2%</td>
</tr>
<tr>
<td>AES with CM on the last two rounds</td>
<td>+ 18.6%</td>
<td>+ 282.5%</td>
</tr>
</tbody>
</table>

**→ High overhead cost,**

but **comparable** to the overhead of usual redundancy approaches

---

**Formal verification of a software countermeasure against instruction skip fault attacks**

I. Introduction

II. Conception of a fault injection bench

III. Validation of a fault model at assembly level

IV. Definition and verification of a software countermeasure

V. Test and experimental evaluation of the countermeasure

VI. Conclusion and perspectives
Countermeasure against a simplified model of attacker (skip of an assembly instruction)

Does not protect against the faults on the data flow

Can be complemented with

- a fault detection countermeasure
- that also protects data loads

Countermeasures against fault attacks on software implemented AES
A. Barenghi, L. Breveglieri, I. Koren, G. Pelosi, F. Regazzoni – WESS 2010, New-York, USA
• Detection of **single faults**
  Instruction skip, some replacements, fault on the data flow

• Proposed for a **restricted set of instructions**
  Arithmetic and logic, load-store
  … but not branches, stack manipulation or flags use

• **High** overhead
  In registers, code size and number of cycles

```assembly
| ldr    | r0, [pc, #34] |
| ldr    | r1, [pc, #38] |
| cmp    | r0, r1       |
| bne    | error        |
```
Study of the impact of the countermeasures for:

- some **isolated instructions**
- some **complex codes**

Chosen isolated instructions:

### Fault tolerance countermeasure

- **bl instruction**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>adr</code> r12, return_label</td>
<td></td>
</tr>
<tr>
<td><code>adr</code> r12, return_label</td>
<td></td>
</tr>
<tr>
<td><code>add</code> lr, r12, #1</td>
<td></td>
</tr>
<tr>
<td><code>add</code> lr, r12, #1</td>
<td></td>
</tr>
<tr>
<td><code>b</code> function</td>
<td></td>
</tr>
<tr>
<td><code>b</code> function</td>
<td></td>
</tr>
</tbody>
</table>

### Fault detection countermeasure

- **ldr instruction**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ldr</code> r0, [pc, #40]</td>
<td></td>
</tr>
<tr>
<td><code>ldr</code> r1, [pc, #38]</td>
<td></td>
</tr>
<tr>
<td><code>cmp</code> r0, r1</td>
<td></td>
</tr>
<tr>
<td><code>bne</code> error</td>
<td></td>
</tr>
</tbody>
</table>
For both countermeasures:

- It is necessary to force a 32-bit encoding

For the tolerance countermeasure:

- On a subroutine call, 97% reduction of the output faults

For the detection countermeasure:

- On a data load, 98% reduction of the output faults
EVALUATION ON COMPLEX CODES

Evaluated on a FreeRTOS-MPU implementation

<table>
<thead>
<tr>
<th>Tolerance CM</th>
<th>Function that changes the privilege level</th>
</tr>
</thead>
<tbody>
<tr>
<td>Detection CM</td>
<td>Function that initializes a task and sets its priority</td>
</tr>
</tbody>
</table>

For the *tolérance countermeasure*:
- → 26% reduction of the faults in the output register
- → Effect that is probably more complex than an instruction skip

For the *detection countermeasure*:
- → 98% reduction of the faults in the output register

Experimental evaluation of two software countermeasures against fault attacks
N. Moro, K. Heydemann, A. Dehbaoui, B. Robisson, E. Encrenaz – IEEE HOST 2014, Arglinton, USA
Combination of the two countermeasures

Both countermeasures are used to reinforce the same code (an AES addRoundKey function)

- Detection CM: more efficient but cannot be applied everywhere
- With proposed CM: protection of the other instructions

90% reduction for the faults, among the remaining ones no usable fault for a cryptanalysis
OUTLINE OF THE PRESENTATION

I. Introduction

II. Conception of a fault injection bench

III. Validation of a fault model at assembly level

IV. Definition and verification of a software countermeasure

V. Test and experimental evaluation of the countermeasure

VI. Conclusion and perspectives
CONCLUSION

• An **accurate fault model** at assembly level
  • Corruption of the transfers from the Flash memory
  • Double-faults possible under certain conditions
  • High percentage of instruction skips

• A **countermeasure tolerant to an instruction skip**
  • Local redundancy at instruction-scale
  • Formally verified with model-checking tools
  • Automatically applicable to a generic code
  • Reinforces especially branches and stack
CONCLUSION

• An experimental evaluation of two countermeasures
  • With another fault detection countermeasure
  • Importance of the encoding of instructions
  • Combination of the two CM very efficient on an AES code
  • First tests with DPA-like side-channel analysis techniques

➔ The proposed fault tolerance countermeasure is a good complement for the detection countermeasure for the subroutine calls and the instructions for which it does not apply
➢ Improvement of the accuracy of fault models
  • Towards a better understanding of instruction replacements
  • Investigate the effects in the different levels of the pipeline
  • Could enable to improve the definition of countermeasures

➢ Test of those countermeasures with side-channel analysis
  • Do they introduce vulnerabilities?
  • How to combine those countermeasures with countermeasures against side-channel analysis?

➢ Automatic application of the countermeasures by the compiler
  • Generate reinforced code for some specified functions
DIFFUSION OF THE RESULTS OF THIS THESIS

• 4 articles in conferences with proceedings
  • COSADE 2013
  • FDTC 2013
  • IEEE HOST 2014
  • IFIP/IEEE VLSI-SoC 2014

• 1 article in a workshop without proceedings
  • PROOFS 2013

• 1 article in a peer-reviewed journal
  • Journal of Cryptographic Engineering 2014

• 3 communications in workshops without proceedings
  • Crypto’Puces 2013
  • Chip-To-Cloud Security Forum 2013
  • TRUDEVICE Workshop 2014
Any questions?

Download the presentation

PDF file