Variable-Length Instruction Set: Feature or Bug? JAIF 2022

Ihab Alshaer, Brice Colombier, Christophe Deleuze, Vincent Beroulle & Paolo Maistri

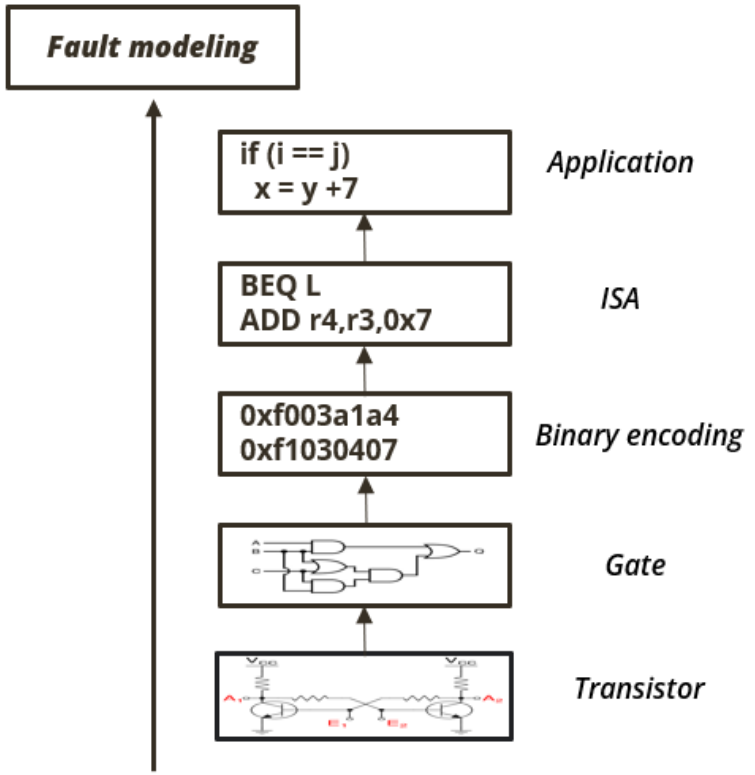
Variable-Length Instruction Set: Feature or Bug?. 25th Euromicro Conference on Digital System Design (DSD 2022), Aug 2022, Maspalomas, Spain.

November 9, 2022, Valence.

Introduction

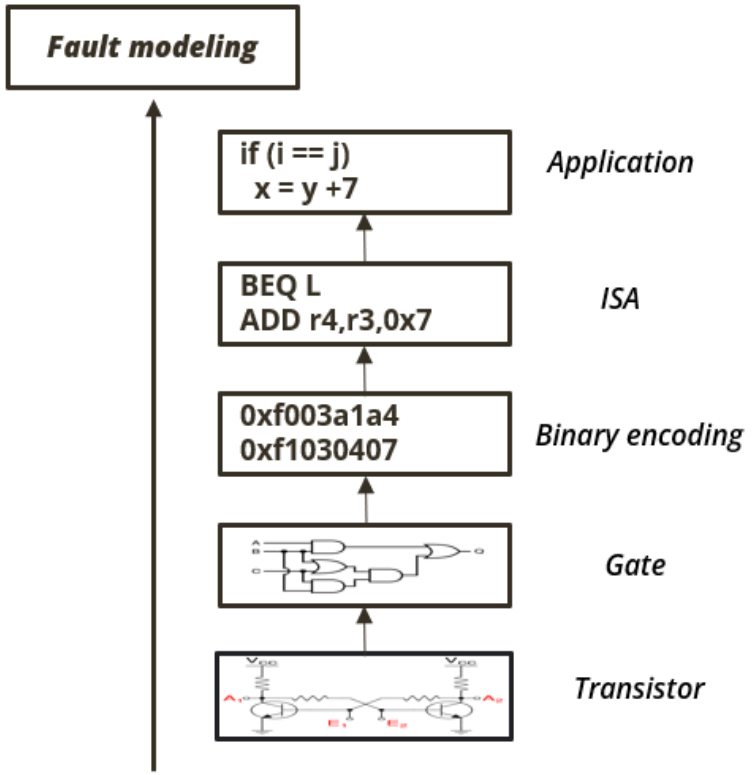
Problematic and motivation

- Secure digital systems against fault attacks requires proper characterizations to build fault models.
- Fault models are necessary:
  - perform vulnerability analysis,
  - design countermeasures.
- Improper fault effect characterization:
  - incomplete fault models, thus non-optimal countermeasures.
    - Over-protections or under-protections



Problematic and motivation

- Secure digital systems against fault attacks requires proper characterizations to build fault models.
- Fault models are necessary:
  - perform vulnerability analysis,
  - design countermeasures.
- Improper fault effect characterization:
  - incomplete fault models, thus non-optimal countermeasures.
    - Over-protections or under-protections



- Previous works focus on ISA level without considering the knowledge of variable-length instructions.
- Many obtained faulty behaviors in the literature remained unexplained.

Variable-length instruction set

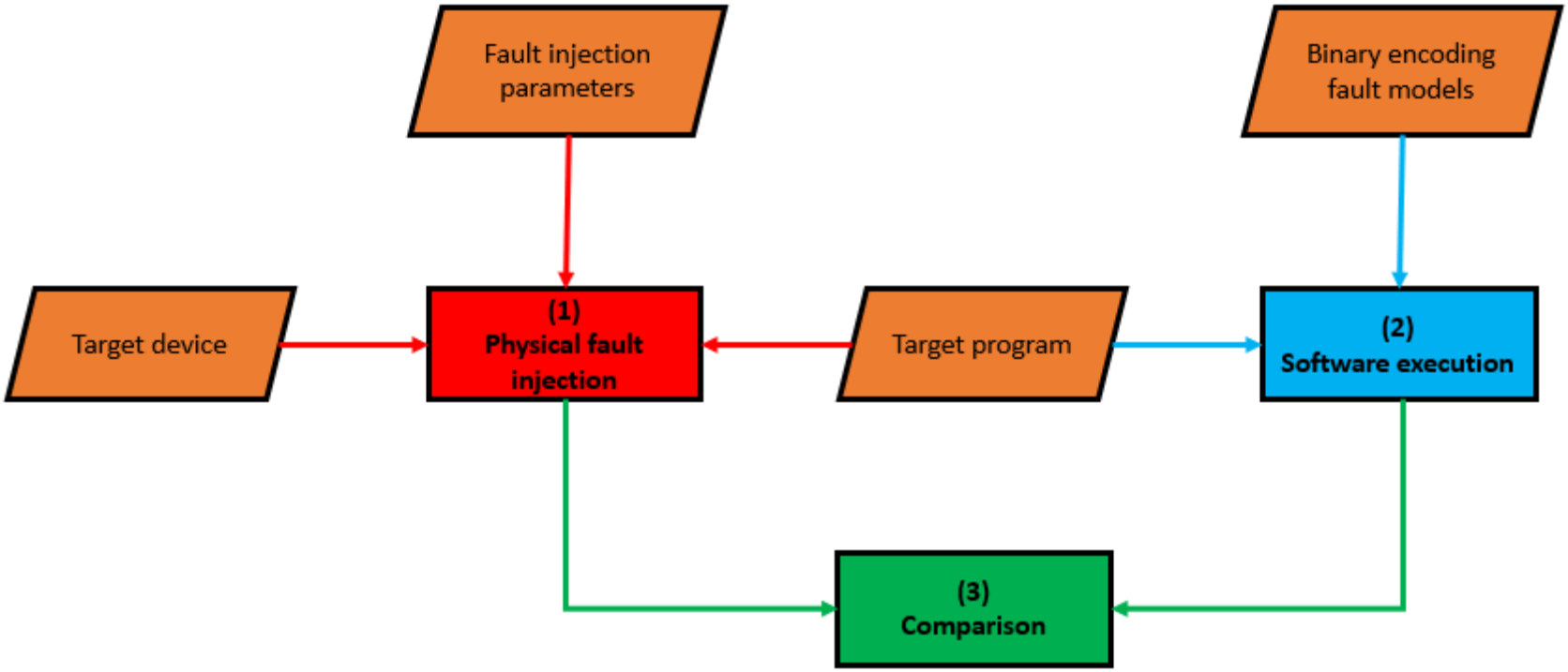
- Has not the same length of encoding for all the instructions.
- Examples:
  - x86: between 1 and 15 bytes.
  - microMIPS: 16 and 32 bits.
  - RISC-V compressed (RVC): 16 and 32 bits.
  - ARM Thumb2: 16 and 32 bits.
- Main advantage: reducing the code size to achieve higher code density.

Main contributions

- Present two new inferred fault models at the binary encoding level:
  - Skip 32 bits
  - Skip & repeat 32 bits.
- Explain a wide range of the obtained faulty behaviors.
- Provide proper characterization for the effects of the observed faulty behaviors at the ISA level.
- Show how the faulty behaviors differ depending on the alignment of the code in memory.
- Provide various examples to violate a predefined security property by exploiting the obtained results.

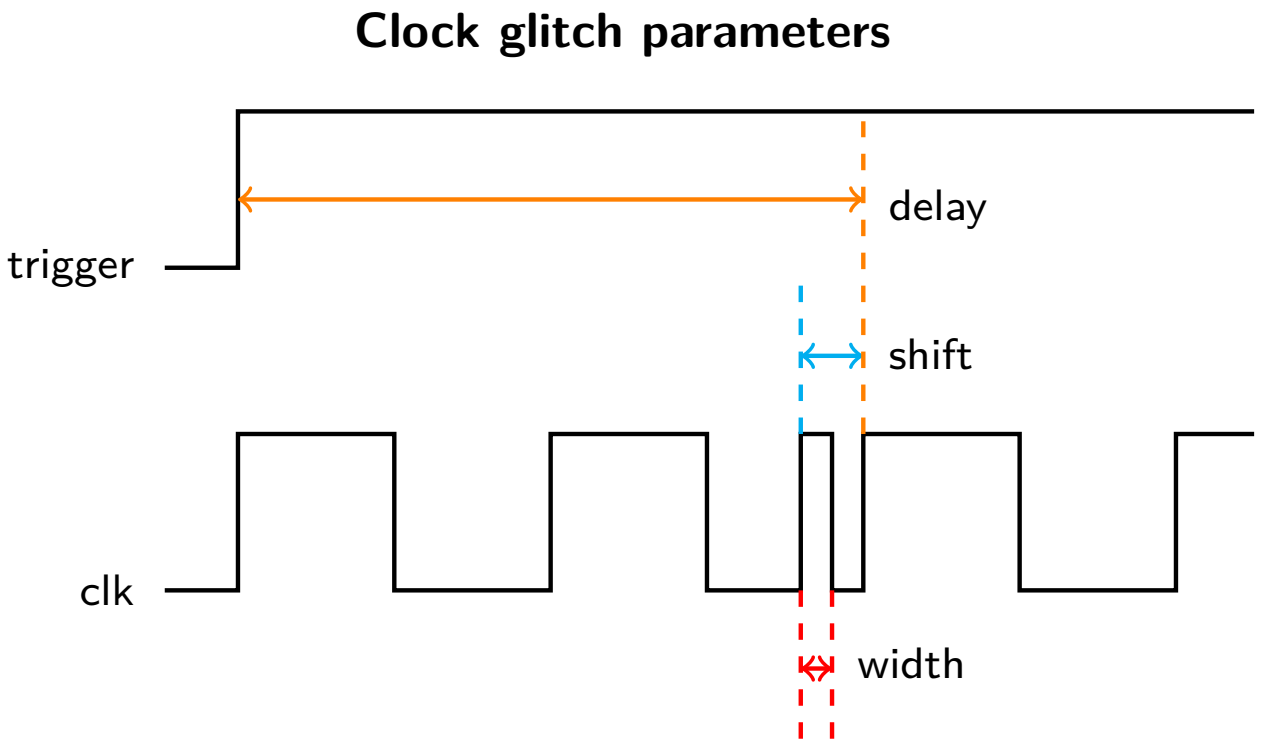
Experimental setup

Overview



- Binary encoding fault models are skip 32 bits and skip & repeat 32 bits.
- The objectives of the comparison are to:
  - explain the observed faulty behaviors,
  - provide proper description of the fault effects at the ISA level.
  - Hence, to propose realistic fault models at the ISA level.

Clock glitch fault injection



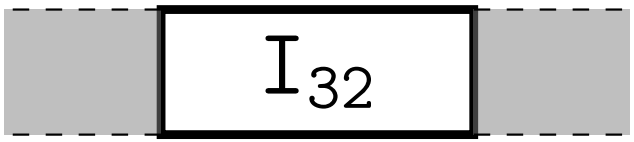
- ChipWhisperer environment is used to perform the clock glitch fault injection.

Target device I

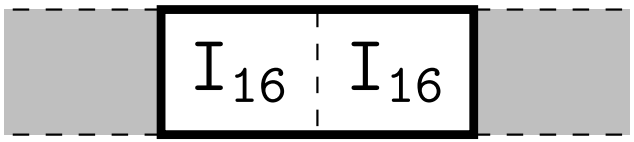
- 32-bit micro-controller.
- Embeds ARM Cortex-M3 processor.
- Supports Thumb2 instruction set:
  - 16- and 32-bit instructions.
- The flash memory access size through the bus to the core is fixed and equals 32 bits.
- The instructions are either aligned or misaligned in the flash memory.

Target device II

Fetching aligned instructions



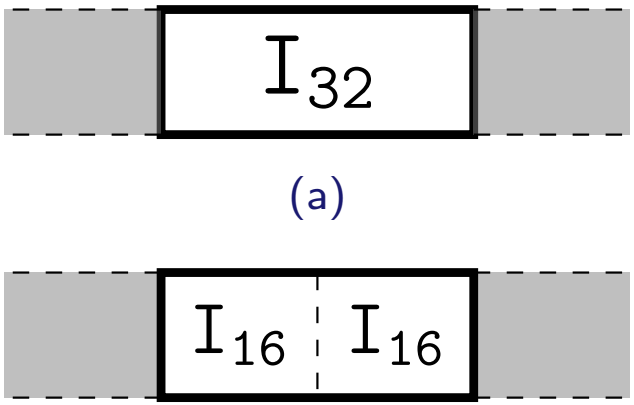
(a)



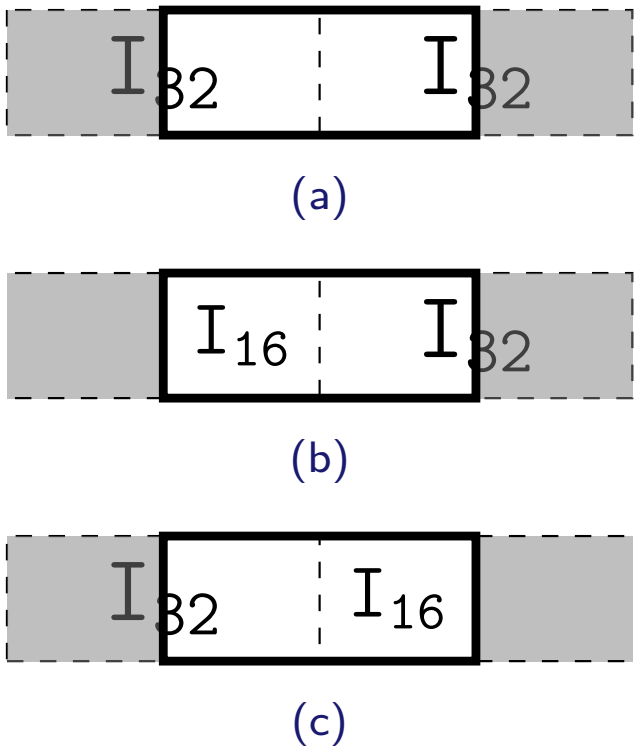
(b)

Target device II

Fetching aligned instructions



Fetching misaligned instructions



Target programs

```
1 MOV R8, R4
2 LSLS R2, R0, 0x10
3 ADD R1, R1, 0x6
4 ADD R3, R3, 0xa
5 ADD R4, R4, 0xb
6 ADD R5, R6, R3
7 ADD R3, R3, 0xf
```

Aligned code target program.

```
1 MOV R8, R4
2 LSLS R2, R0, 0x10
3 ADD R1, R1, 0x6
4 ADD R3, R3, 0xa
5 ADD R4, R4, 0xb
6 ADD R5, R6, R3
7 ADD R3, R3, 0xf
```

Misaligned code target program.

- MOV and LSLS are 16-bit instructions.
- All the **ADD** are 32-bit instructions.

Experimental results and analysis

Aligned code scenario I

```
1 46a00402
2 f1010106
3 f103030a
4 f104040b
5 eb060503
6 f103030f
```

Binary encoding of the aligned code in hex. format.

```
1 MOV R8, R4
2 LSLS R2, R0, 0x10
3 ADD R1, R1, 0x6
4 ADD R3, R3, 0xa
5 ADD R4, R4, 0xb
6 ADD R5, R6, R3
7 ADD R3, R3, 0xf
```

Aligned code target program.

- Skip 32 bits: the 32 bits at line *i* are skipped, and the execution resumes from line *i+1*.
- Skip & repeat 32 bits: the 32 bits at line *i+1* are skipped and the 32 bits at line *i* are repeated.

Aligned code scenario II

Skip & repeat 32 bits I

```
1 46a00402
2 f1010106
3 f103030a
4 f104040b
5 eb060503
6 f103030f
```

Original encoding.

Aligned code scenario II

Skip & repeat 32 bits I

```
1 46a00402
2 f1010106
3 f103030a
4 f104040b
5 eb060503
6 f103030f
```

Original encoding.

```
1 46a00402
2 f1010106
3 f1010106
4 f104040b
5 eb060503
6 f103030f
```

Skip line 3 and repeat line 2.

Aligned code scenario II

Skip & repeat 32 bits I

```
1 46a00402
2 f1010106
3 f103030a
4 f104040b
5 eb060503
6 f103030f
```

Original encoding.

```
1 46a00402
2 f1010106
3 f1010106
4 f104040b
5 eb060503
6 f103030f
```

Skip line 3 and repeat line 2.

```
1 MOV R8, R4
2 LSLS R2, R0, 0x10
3 ADD R1, R1, 0x6
4 ADD R1, R1, 0x6
5 ADD R4, R4, 0xb
6 ADD R5, R6, R3
7 ADD R3, R3, 0xf
```

Single instruction skip and single instruction repeat.

```
1 MOV R8, R4
2 LSLS R2, R0, 0x10
3 ADD R1, R1, 0x6
4 ADD R3, R3, 0xa
5 ADD R4, R4, 0xb
6 ADD R5, R6, R3
7 ADD R3, R3, 0xf
```

Original code.

Aligned code scenario III

Skip 32 bits I

```
1 46a00402
2 f1010106
3 f103030a
4 f104040b
5 eb060503
6 f103030f
```

Original encoding.

Aligned code scenario III

Skip 32 bits I

```
1 46a00402
2 f1010106
3 f103030a
4 f104040b
5 eb060503
6 f103030f
```

Original encoding.

```
1 46a00402
2 f1010106
3 f103030a
4 ~~f104040b~~
5 eb060503
6 f103030f
```

Skip line 4.

Aligned code scenario III

Skip 32 bits I

```
1 46a00402
2 f1010106
3 f103030a
4 f104040b
5 eb060503
6 f103030f
```

Original encoding.

```
1 46a00402
2 f1010106
3 f103030a
4 f104040b
5 eb060503
6 f103030f
```

Skip line 4.

```
1 MOV R8, R4
2 LSLS R2, R0, 0x10
3 ADD R1, R1, 0x6
4 ADD R3, R3, 0xa
5 ADD R5, R6, R3
6 ADD R3, R3, 0xf
```

Single instruction skip.

```
1 MOV R8, R4
2 LSLS R2, R0, 0x10
3 ADD R1, R1, 0x6
4 ADD R3, R3, 0xa
5 ADD R4, R4, 0xb
6 ADD R5, R6, R3
7 ADD R3, R3, 0xf
```

Original code.

Aligned code scenario IV

Skip 32 bits II

```
1 46a00402
2 f1010106
3 f103030a
4 f104040b
5 eb060503
6 f103030f
```

Original encoding.

Aligned code scenario IV

Skip 32 bits II

```
1 46a00402
2 f1010106
3 f103030a
4 f104040b
5 eb060503
6 f103030f
```

Original encoding.

```
1 ~~46a00402~~
2 f1010106
3 f103030a
4 f104040b
5 eb060503
6 f103030f
```

Skip line 1.

Aligned code scenario IV

Skip 32 bits II

```
1 46a00402
2 f1010106
3 f103030a
4 f104040b
5 eb060503
6 f103030f
```

Original encoding.

```
1 ~~46a00402~~
2 f1010106
3 f103030a
4 f104040b
5 eb060503
6 f103030f
```

Skip line 1.

```
1 ADD R1, R1, 0x6
2 ADD R3, R3, 0xa
3 ADD R4, R4, 0xb
4 ADD R5, R6, R3
5 ADD R3, R3, 0xf
```

Double instruction skip.

```
1 MOV R8, R4
2 LSLS R2, R0, 0x10
3 ADD R1, R1, 0x6
4 ADD R3, R3, 0xa
5 ADD R4, R4, 0xb
6 ADD R5, R6, R3
7 ADD R3, R3, 0xf
```

Original code.

Misaligned code scenario I

```
1 0402f101
2 0106f103
3 030af104
4 040beb06
5 0503f103
6 030fbf00 // bf00: NOP.
```

Binary encoding of the misaligned code in hex. format.

```
1 LSLS R2, R0, 0x10
2 ADD R1, R1, 0x6
3 ADD R3, R3, 0xa
4 ADD R4, R4, 0xb
5 ADD R5, R6, R3
6 ADD R3, R3, 0xf
```

Misaligned code target program.

- Skip 32 bits: the 32 bits at line *i* are skipped, and the execution resumes from line *i+1*.
- Skip & repeat 32 bits: the 32 bits at line *i+1* are skipped and the 32 bits at line *i* are repeated.

Misaligned code scenario II

Skip & repeat 32 bits I

```
1 0402f101
2 0106f103
3 030af104
4 040beb06
5 0503f103
6 030fbf00
```

Original encoding.

Misaligned code scenario II

Skip & repeat 32 bits I

```
1 0402f101
2 0106f103
3 030af104
4 040beb06
5 0503f103
6 030fbf00
```

Original encoding.

```
1 0402f101
2 0106f103
3 030a f104
4 030af104
5 0503 f103
6 030fbf00
```

Skip line 4 & repeat line 3.

Misaligned code scenario II

Skip & repeat 32 bits I

```
1 0402f101
2 0106f103
3 030af104
4 040beb06
5 0503f103
6 030fbf00
```

Original encoding.

```
1 0402f101
2 0106f103
3 030a f104
4 030af104
5 0503 f103
6 030fbf00
```

Skip line 4 & repeat line 3.

```
1 LSLS R2, R0, 0x10
2 ADD R1, R1, 0x6
3 ADD R3, R3, 0xa
4 ADD R3, R4, 0xa
5 ADD R5, R4, 0x3
6 ADD R3, R3, 0xf
```

Double instruction corruption.

```
1 LSLS R2, R0, 0x10
2 ADD R1, R1, 0x6
3 ADD R3, R3, 0xa
4 ADD R4, R4, 0xb
5 ADD R5, R6, R3
6 ADD R3, R3, 0xf
```

Original code.

Misaligned code scenario III

Skip 32 bits I

```
1 0402f101
2 0106f103
3 030af104
4 040beb06
5 0503f103
6 030fbf00
```

Original encoding.

Misaligned code scenario III

Skip 32 bits I

```
1 0402f101
2 0106f103
3 030af104
4 040beb06
5 0503f103
6 030fbf00
```

Original encoding.

```
1 0402f101
2 0106 f103
3 030af104
4  040b eb06
5 0503f103
6 030fbf00
```

Skip line 3.

Misaligned code scenario III

Skip 32 bits I

```
1 0402f101
2 0106f103
3 030af104
4 040beb06
5 0503f103
6 030fbf00
```

Original encoding.

```
1 0402f101
2 0106 f103
3 030af104
4 040b eb06
5 0503f103
6 030fbf00
```

Skip line 3.

```
1 LSLS R2, R0, 0x10
2 ADD R1, R1, 0x6
3 ADD R4, R3, 0xb
4 ADD R5, R6, R3
5 ADD R3, R3, 0xf
```

Single instruction skip and single instruction corruption.

```
1 LSLS R2, R0, 0x10
2 ADD R1, R1, 0x6
3 ADD R3, R3, 0xa
4 ADD R4, R4, 0xb
5 ADD R5, R6, R3
6 ADD R3, R3, 0xf
```

Original code.

Misaligned code scenario IV

Skip 32 bits II

```
1 0402f101
2 0106f103
3 030af104
4 040beb06
5 0503f103
6 030fbf00
```

Original encoding.

Misaligned code scenario IV

Skip 32 bits II

```
1 0402f101
2 0106f103
3 030af104
4 040beb06
5 0503f103
6 030fbf00
```

Original encoding.

```
1 0402f101
2 0106 f103
3 030af104
4 040beb06
5 0503f103
6 030fbf00
```

Skip line 1.

Misaligned code scenario IV

Skip 32 bits II

```
1 0402f101
2 0106f103
3 030af104
4 040beb06
5 0503f103
6 030fbf00
```

Original encoding.

```
1 ~~0402f101~~
2 0106 f103
3 030af104
4 040beb06
5 0503f103
6 030fbf00
```

Skip line 1.

```
1 LSLS R6, R0, 0x4
2 ADD R3, R3, 0xa
3 ADD R4, R4, 0xb
4 ADD R5, R6, R3
5 ADD R3, R3, 0xf
```

Double instruction skip and new instruction execution.

```
1 LSLS R2, R0, 0x10
2 ADD R1, R1, 0x6
3 ADD R3, R3, 0xa
4 ADD R4, R4, 0xb
5 ADD R5, R6, R3
6 ADD R3, R3, 0xf
```

Original code.

Misaligned code scenario IV

Skip 32 bits II

```
1 0402f101
2 0106f103
3 030af104
4 040beb06
5 0503f103
6 030fbf00
```

Original encoding.

```
1 ~~0402f101~~
2 0106 f103
3 030af104
4 040beb06
5 0503f103
6 030fbf00
```

Skip line 1.

```
1 LSLS R6, R0, 0x4
2 ADD R3, R3, 0xa
3 ADD R4, R4, 0xb
4 ADD R5, R6, R3
5 ADD R3, R3, 0xf
```

Double instruction skip and new instruction execution.

```
1 LSLS R2, R0, 0x10
2 ADD R1, R1, 0x6
3 ADD R3, R3, 0xa
4 ADD R4, R4, 0xb
5 ADD R5, R6, R3
6 ADD R3, R3, 0xf
```

Original code.

- Changing the destination register (R1) and/or the immediate value (0x6) allows observing the execution of other new instructions.

Misaligned code scenario V

More on the ability of executing a new instruction

Original instruction	Least-significant 16 bits	New instruction
`ADD R4, R1, 0x9`	`0x0409`	`LSLS R1, R1, 0x10`
`ADD R0, R1, 0x46c`	`0x406c`	`EORS R4, R5`
`ADD R12, R1, 0x60c`	`0x6c0c`	`LDR R4, [R1, 0x40]`
`ADD R0, R1, 0x161`	`0x1061`	`ASRS R1, R4, 0x1`
`ADD R0, R1, 0x205`	`0x2005`	`MOV R0, 0x5`
`ADD R3, R1, 0x416`	`0x4316`	`ORRS R6, R2`

Table: Effect of last observed behavior with different destination register and/or immediate value.

Exploitation

Exploit the ability of executing a new instruction

- The security property is to not modify the program counter to a specific address stored in **R8**.
- This security property is violated if we manage to execute the **MOV PC, R8** instruction (**0x46c7**).

```
1  //R8 = address of line 10
2  //series of NOPs
3  LSLS R2, R0, 0x10
4  //any instruction from the Table
5  ADD R3, R3, 0xa
6  ADD R4, R4, 0xb
7  ADD R5, R6, R3
8  ADD R3, R3, 0xf
9  //series of NOPs
10 LDR R1, [R1,0xf00]
11 MOV R9, R6
```

Target program for exploitation example.

Original instruction	Least-significant 16 bits
`ADD R6, R1, 0x4c7`	0x46c7
`SUB R6, R1, 0x4c7`	0x46c7
`MOVW R6, 0x4c7`	0x46c7
`LDR R4, [r0, 0x6c7]`	0x46c7
`ORR R6, R6, 0x63800000`	0x46c7

Table: Instructions that lead to modify the PC to the value in R8 when performing clock glitch fault injection.

Conclusions and future works

Conclusions and Future works

- The observed faulty behaviors at ISA level can dramatically change depending on the code alignment in memory.
  - Thumb2 instruction set supports variable-length instructions.
- The observed behaviors can be explained at the binary encoding level with the two fault models: skip 32 bits and skip & repeat 32 bits.
- Proper characterization and realistic ISA fault models are provided.
- The provided description clearly explains many faulty behaviors mentioned in the literature.
- Show how such behaviors can be exploited.

Conclusions and Future works

- The observed faulty behaviors at ISA level can dramatically change depending on the code alignment in memory.
  - Thumb2 instruction set supports variable-length instructions.
- The observed behaviors can be explained at the binary encoding level with the two fault models: skip 32 bits and skip & repeat 32 bits.
- Proper characterization and realistic ISA fault models are provided.
- The provided description clearly explains many faulty behaviors mentioned in the literature.
- Show how such behaviors can be exploited.

- Exploit the results in a real-life application.
- Looking for countermeasures at different system levels.
- Target other architectures that support variable-length instruction sets.

Thank you! Questions?

Ihab Alshaer, Brice Colombier, Christophe Deleuze, Vincent Beroulle & Paolo Maistri

{first.last}@univ-grenoble-alpes.fr