Controlling PC on ARM using Fault Injection

Niek Timmers
timmers@riscure.com

Albert Spruyt
spruyt@riscure.com

Marc Witteman
witteman@riscure.com

August 16, 2016
Table of contents

1. Attack strategy
2. Practical attack scenario
3. Simulation
4. Experimentation
5. Countermeasures
6. Conclusion
Fault injection techniques

clock  voltage  electromagnetic  laser
## Fault injection fault model

### Instruction corruption

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Binary Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV R0, R1</td>
<td>1110000110100000000000000000000000000000010</td>
</tr>
<tr>
<td>MOV R0, R2</td>
<td>11100001101000000000000000000001</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Binary Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV R0, R1</td>
<td>11100001101000000000000000000001010111000000100000001</td>
</tr>
<tr>
<td>STR R7, [R7, #16]</td>
<td>11100101100010111011000000010000000100000000000000</td>
</tr>
</tbody>
</table>

### Instruction skipping

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Binary Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV R0, R1</td>
<td>11100001101000000000000000000001</td>
</tr>
<tr>
<td>MOV R1, R1</td>
<td>1110000110100000000000000000000100000000001</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Binary Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV R0, R1</td>
<td>11100001101000000000000000000001</td>
</tr>
<tr>
<td>MOV R1, R1</td>
<td>1110000110100000000000000000000100000000001</td>
</tr>
<tr>
<td>MOV R6, R6</td>
<td>11100001101000000000000000000001110000000000110</td>
</tr>
</tbody>
</table>
## Fault injection fault model

### Instruction corruption

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Binary Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV R0, R1</td>
<td>11100001110101000000000000000000000010</td>
</tr>
<tr>
<td>MOV R0, R2</td>
<td>11100001110101000000000000000000000010</td>
</tr>
<tr>
<td>MOV R0, R1</td>
<td>11100001110101000000000000000000000010</td>
</tr>
<tr>
<td>STR R7, [R7, #16]</td>
<td>11100101100010010111000000000100000</td>
</tr>
</tbody>
</table>

### Instruction skipping

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Binary Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV R0, R1</td>
<td>11100001110101000000000000000000000010</td>
</tr>
<tr>
<td>MOV R1, R1</td>
<td>11100001110101000000000000000000000010</td>
</tr>
<tr>
<td>MOV R6, R6</td>
<td>111000011101010000000000000000000000110</td>
</tr>
<tr>
<td>MOV R0, R1</td>
<td>11100001110101000000000000000000000010</td>
</tr>
<tr>
<td>MOV R6, R6</td>
<td>111000011101010000000000000000000000110</td>
</tr>
</tbody>
</table>
## Fault injection fault model

### Instruction corruption

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Binary Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV R0, R1</td>
<td>11100001110101000000000000000000000000001</td>
</tr>
<tr>
<td>MOV R0, R2</td>
<td>111000011101010000000000000000001</td>
</tr>
<tr>
<td>MOV R7, [R7, #16]</td>
<td>1110010110001010111000000010000</td>
</tr>
</tbody>
</table>

### Instruction skipping

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Binary Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV R0, R1</td>
<td>11100001110101000000000000000000000000001</td>
</tr>
<tr>
<td>MOV R1, R1</td>
<td>111000011101010000000000000000001</td>
</tr>
<tr>
<td>MOV R6, R6</td>
<td>11100001110100000000110000000000110</td>
</tr>
</tbody>
</table>
Fault injection fault model

**Instruction corruption**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Binary Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV R0, R1</td>
<td>1110000111010000000000000000000000</td>
</tr>
<tr>
<td>MOV R0, R2</td>
<td>11100001110100000000000000000010</td>
</tr>
<tr>
<td>MOV R0, R1</td>
<td>111000011101000000000000000000010</td>
</tr>
<tr>
<td>STR R7, [R7, #16]</td>
<td>111001011000101011100000010000</td>
</tr>
</tbody>
</table>

**Instruction skipping**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Binary Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV R0, R1</td>
<td>111000011101000000000000000000001</td>
</tr>
<tr>
<td>MOV R1, R1</td>
<td>111000011101000000000000000000001</td>
</tr>
<tr>
<td>MOV R0, R1</td>
<td>111000011101000000000000000000001</td>
</tr>
<tr>
<td>MOV R6, R6</td>
<td>1110000111010000000000000110000000110</td>
</tr>
</tbody>
</table>
**Fault injection fault model**

### Instruction corruption

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV R0, R1</td>
<td>111000011010000000000000000000000</td>
</tr>
<tr>
<td>MOV R0, R2</td>
<td>11100001101000000000000000000010</td>
</tr>
<tr>
<td>MOV R0, R1</td>
<td>11100001101000000000000000000001</td>
</tr>
<tr>
<td>STR R7, [R7, #16]</td>
<td>1110010110001001011100000010000</td>
</tr>
</tbody>
</table>

### Instruction skipping

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV R0, R1</td>
<td>11100001101000000000000000000001</td>
</tr>
<tr>
<td>MOV R1, R1</td>
<td>11100001101000000000000000000001</td>
</tr>
<tr>
<td>MOV R0, R1</td>
<td>11100001101000000000000000000001</td>
</tr>
<tr>
<td>MOV R6, R6</td>
<td>111000011010000000000000011000000110</td>
</tr>
</tbody>
</table>
Fault injection fault model

**Instruction corruption**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Binary Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV R0, R1</td>
<td>1110000011010000000000000000000000000000000000010</td>
</tr>
<tr>
<td>MOV R0, R2</td>
<td>1110000011010000000000000000000000000000000000001</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Binary Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV R0, R1</td>
<td>1110000011010000000000000000000000000000000000001</td>
</tr>
<tr>
<td>STR R7, [R7, #16]</td>
<td>111001011000000001110000000000000000000000000001</td>
</tr>
</tbody>
</table>

**Instruction skipping**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Binary Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV R0, R1</td>
<td>1110000011010000000000000000000000000000000000001</td>
</tr>
<tr>
<td>MOV R1, R1</td>
<td>1110000011010000000000000000000000000000000000001</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Binary Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV R0, R1</td>
<td>1110000011010000000000000000000000000000000000001</td>
</tr>
<tr>
<td>MOV R6, R6</td>
<td>111000001101000000000000000000000000000000000000110</td>
</tr>
</tbody>
</table>
故障注入故障模型

**Instruction corruption**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Binary Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV R0, R1</td>
<td>1110000111010000000000000000000000010</td>
</tr>
<tr>
<td>MOV R0, R2</td>
<td>1110000111010000000000000000000010</td>
</tr>
<tr>
<td>MOV R0, R1</td>
<td>111000011101000000000000000000001</td>
</tr>
<tr>
<td>STR R7, [R7, #16]</td>
<td>1110010110001001110000000010000</td>
</tr>
</tbody>
</table>

**Instruction skipping**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Binary Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV R0, R1</td>
<td>111000011101000000000000000000000001</td>
</tr>
<tr>
<td>MOV R1, R1</td>
<td>11100001110100000000000000000000001</td>
</tr>
<tr>
<td>MOV R0, R1</td>
<td>11100001110100000000000000000000001</td>
</tr>
<tr>
<td>MOV R6, R6</td>
<td>111000011101000000000000000000000100010</td>
</tr>
</tbody>
</table>
Why ARM?

- ARM is everywhere

- The PC register is directly accessible in ARM (AArch32)
Why ARM?

- ARM is everywhere

- The PC register is directly accessible in ARM (AArch32)
## Why ARM?

- ARM is everywhere

<table>
<thead>
<tr>
<th>Application</th>
<th>Chip Function</th>
<th>2015</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Mobile Computing</strong></td>
<td>Apps Processors</td>
<td>1,800</td>
<td>1,800</td>
<td>1,600</td>
<td>&gt;85%</td>
</tr>
<tr>
<td></td>
<td>Connectivity and Control</td>
<td>11,000</td>
<td>4,000</td>
<td>37%</td>
<td></td>
</tr>
<tr>
<td><strong>Consumer Electronics</strong></td>
<td>Apps Processors</td>
<td>3,600</td>
<td>1,000</td>
<td>700</td>
<td>70%</td>
</tr>
<tr>
<td></td>
<td>Connectivity and Control</td>
<td>8,000</td>
<td>3,000</td>
<td>40%</td>
<td></td>
</tr>
<tr>
<td><strong>Enterprise Infrastructure</strong></td>
<td>Servers</td>
<td>300</td>
<td>22</td>
<td>&gt;0</td>
<td>&lt;1%</td>
</tr>
<tr>
<td></td>
<td>Networking - Infrastructure</td>
<td>140</td>
<td>20</td>
<td>15%</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Networking - Home and Office</td>
<td>700</td>
<td>200</td>
<td>30%</td>
<td></td>
</tr>
<tr>
<td><strong>Automotive</strong></td>
<td>Apps Processors</td>
<td>90</td>
<td>68</td>
<td>65</td>
<td>&gt;95%</td>
</tr>
<tr>
<td></td>
<td>Control</td>
<td>2,700</td>
<td>200</td>
<td>7%</td>
<td></td>
</tr>
<tr>
<td><strong>Embedded Intelligence</strong></td>
<td>Apps Processors</td>
<td>500</td>
<td>350</td>
<td>70%</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Connectivity</td>
<td>600</td>
<td>300</td>
<td>50%</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Control</td>
<td>20,000</td>
<td>4,400</td>
<td>22%</td>
<td></td>
</tr>
<tr>
<td><strong>Total (in millions)</strong></td>
<td></td>
<td>46,500</td>
<td>14,800</td>
<td>32%</td>
<td></td>
</tr>
</tbody>
</table>

- The PC register is directly accessible in ARM (AArch32)
Why ARM?

- ARM is everywhere

<table>
<thead>
<tr>
<th>Application</th>
<th>Chip Function</th>
<th>2015</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Device Shipments</td>
<td>Chip Shipments</td>
</tr>
<tr>
<td>Mobile Computing *</td>
<td>Apps Processors</td>
<td>1,800</td>
</tr>
<tr>
<td></td>
<td>Connectivity and Control</td>
<td>11,000</td>
</tr>
<tr>
<td>Consumer Electronics **</td>
<td>Apps Processors</td>
<td>3,600</td>
</tr>
<tr>
<td></td>
<td>Connectivity and Control</td>
<td>8,000</td>
</tr>
<tr>
<td>Enterprise Infrastructure</td>
<td>Servers</td>
<td>300</td>
</tr>
<tr>
<td></td>
<td>Networking - Infrastructure</td>
<td>140</td>
</tr>
<tr>
<td></td>
<td>Networking - Home and Office</td>
<td>700</td>
</tr>
<tr>
<td>Automotive</td>
<td>Apps Processors</td>
<td>90</td>
</tr>
<tr>
<td></td>
<td>Control</td>
<td>2,700</td>
</tr>
<tr>
<td>Embedded Intelligence</td>
<td>Apps Processors</td>
<td>500</td>
</tr>
<tr>
<td></td>
<td>Connectivity</td>
<td>600</td>
</tr>
<tr>
<td></td>
<td>Control</td>
<td>20,000</td>
</tr>
<tr>
<td>Total (in millions)</td>
<td>46,500</td>
<td>14,800</td>
</tr>
</tbody>
</table>

- The PC register is directly accessible in ARM (AArch32)
## Why ARM?

- ARM is everywhere

<table>
<thead>
<tr>
<th>Application</th>
<th>Chip Function</th>
<th>2015</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Device Shipments</td>
</tr>
<tr>
<td>Mobile Computing *</td>
<td>Apps Processors</td>
<td>1,800</td>
</tr>
<tr>
<td></td>
<td>Connectivity and Control</td>
<td>11,000</td>
</tr>
<tr>
<td>Consumer Electronics **</td>
<td>Apps Processors</td>
<td>3,600</td>
</tr>
<tr>
<td></td>
<td>Connectivity and Control</td>
<td>8,000</td>
</tr>
<tr>
<td>Enterprise Infrastructure</td>
<td>Servers</td>
<td>300</td>
</tr>
<tr>
<td></td>
<td>Networking - Infrastructure</td>
<td>140</td>
</tr>
<tr>
<td></td>
<td>Networking - Home and Office</td>
<td>700</td>
</tr>
<tr>
<td>Automotive</td>
<td>Apps Processors</td>
<td>90</td>
</tr>
<tr>
<td></td>
<td>Control</td>
<td>2,700</td>
</tr>
<tr>
<td>Embedded Intelligence</td>
<td>Apps Processors</td>
<td>500</td>
</tr>
<tr>
<td></td>
<td>Connectivity</td>
<td>600</td>
</tr>
<tr>
<td></td>
<td>Control</td>
<td>20,000</td>
</tr>
<tr>
<td>Total (in millions)</td>
<td></td>
<td>46,500</td>
</tr>
</tbody>
</table>

- The PC register is directly accessible in ARM (AArch32)
All systems copy data from A to B!

Single word copy using LDR / STR

WordCopy:
1
2	LDR r3, [r1], #4
3	STR r3, [r0], #4
4	SUBS r2, r2, #4
5	BGE WordCopy

Multi-word copy using LDMIA / STMIA

MultiWorldCopy:
1
2	LDMIA r1!, {r3 - r10}
3	STMIA r0!, {r3 - r10}
4	SUBS r2, r2, #32
5	BGE MultiWorldCopy
All systems copy data from A to B!

Single word copy using LDR / STR

WordCopy:
1. LDR r3, [r1], #4
2. STR r3, [r0], #4
3. SUBS r2, r2, #4
4. BGE WordCopy

Multi-word copy using LDMIA / STMIA

MultiWorldCopy:
1. LDMIA r1!, {r3 - r10}
2. STMIA r0!, {r3 - r10}
3. SUBS r2, r2, #32
4. BGE MultiWorldCopy
All systems copy data from A to B!

Single word copy using LDR / STR

```
WordCopy:
1       LDR  r3, [r1], #4
2       STR  r3, [r0], #4
3       SUBS r2, r2, #4
4       BGE  WordCopy
```

Multi-word copy using LDMIA / STMIA

```
MultiWorldCopy:
1       LDMIA r1!, {r3 - r10}
2       STMIA r0!, {r3 - r10}
3       SUBS r2, r2, #32
4       BGE  MultiWorldCopy
```
All systems copy data from A to B!

Single word copy using LDR / STR

```assembly
WordCopy:
1 LDR r3, [r1], #4
2 STR r3, [r0], #4
3 SUBS r2, r2, #4
4 BGE WordCopy
```

Multi-word copy using LDMIA / STMIA

```assembly
MultiWorldCopy:
1 LDMIA r1!, {r3 - r10}
2 STMIA r0!, {r3 - r10}
3 SUBS r2, r2, #32
4 BGE MultiWorldCopy
```
All systems copy data from A to B!

**Single word copy using LDR / STR**

1. **WordCopy:**
   1. LDR r3, [r1], #4
   2. STR r3, [r0], #4
   3. SUBS r2, r2, #4
   4. BGE WordCopy

**Multi-word copy using LDMIA / STMIA**

1. **MultiWorldCopy:**
   1. LDMIA r1!, {r3 - r10}
   2. STMIA r0!, {r3 - r10}
   3. SUBS r2, r2, #32
   4. BGE MultiWorldCopy
Why are copy operations interesting?

- They operate on attacker controlled data
- They are executed multiple times consecutively
- They are typically not protected
Why are copy operations interesting?

- They operate on attacker controlled data
- They are executed multiple times consecutively
- They are typically not protected
Why are copy operations interesting?

- They operate on attacker controlled data
- They are executed multiple times consecutively
  - They are typically not protected
Why are copy operations interesting?

- They operate on attacker controlled data
- They are executed multiple times consecutively
- They are typically not protected
Corrupting load instructions to control PC

Controlling PC using LDR

LDR r3, [r1], #4  11100100100100010011000000000100

LDR PC, [r1], #4  1110010010010001111100000000100

Controlling PC using LDMIA

LDMIA r1!,{r3-r10}  1110100010110001000011111111000

LDMIA r1!,{r3-r10,PC}  11101000101100011000011111111000

Important: The destination register(s) is encoded differently!
Corrupting load instructions to control PC

**Controlling PC using LDR**

LDR r3, [r1], #4 11100100100100010011000000000100

LDR PC, [r1], #4 11100100100100011111000000000100

**Controlling PC using LDMIA**

LDMIA r1!, {r3-r10} 1110100010110001000011111111000

LDMIA r1!, {r3-r10, PC} 111010001011000111110000111111111000

*Important:* The destination register(s) is encoded differently!
Corrupting load instructions to control PC

Controlling PC using LDR

LDR r3, [r1], #4
11100100100100010011000000000100

LDR PC, [r1], #4
1110010010010001111100000000100

Controlling PC using LDMIA

LDMIA r1!, {r3-r10}
11101000101100010000011111111000

LDMIA r1!, {r3-r10, PC}
11101000101100011000011111111000

Important: The destination register(s) is encoded differently!
Corrupting load instructions to control PC

**Controlling PC using LDR**

LDR r3, [r1], #4  11100100100100010011000000000100

LDR PC, [r1], #4  11100100100100011111000000000100

**Controlling PC using LDMIA**

LDMIA r1!, {r3-r10}  11101000101100010000011111111000

LDMIA r1!, {r3-r10, PC}  111010001011000111000011111111000

**Important:** The destination register(s) is encoded differently!
Corrupting load instructions to control PC

**Controlling PC using LDR**

```
LDR r3, [r1], #4  1110010010010001001100000000100
```

```
LDR PC, [r1], #4  111001001001000111110000000100
```

**Controlling PC using LDMIA**

```
LDMIA r1!,{r3-r10}  1110100010110001000011111111000
```

```
LDMIA r1!,{r3-r10,PC}  1110100010110001111111111111000
```

*Important:* The destination register(s) is encoded differently!
Corrupting load instructions to control PC

**Controlling PC using LDR**

LDR r3, [r1], #4 11100100100100010011000000000100

LDR PC, [r1], #4 1110010010010001111100000000100

**Controlling PC using LDMIA**

LDMIA r1!,{r3-r10} 11101000101100010000111111111000

LDMIA r1!,{r3-r10,PC} 1110100010110001111111111111000

*Important*: The destination register(s) is encoded differently!
Corrupting load instructions to control PC

**Controlling PC using LDR**

LDR r3, [r1], #4  11100100100100010011000000000100

LDR PC, [r1], #4  11100100100100011111000000000100

**Controlling PC using LDMIA**

LDMIA r1!,{r3-r10}  1110100010110001000011111111000

LDMIA r1!,{r3-r10,PC}  1110100010110001111111111111000

*Important*: The destination register(s) is encoded differently!
## Corrupting load instructions to control PC

### Controlling PC using LDR

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDR r3, [r1], #4</td>
<td>111001001001001001001100000000000100</td>
</tr>
<tr>
<td>LDR PC, [r1], #4</td>
<td>11100100100100100011111111000000000100</td>
</tr>
</tbody>
</table>

### Controlling PC using LDMIA

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDMIA r1!,{r3-r10}</td>
<td>11101000101100010000011111111111000</td>
</tr>
<tr>
<td>LDMIA r1!,{r3-r10,PC}</td>
<td>111010001011000100000111111111111000</td>
</tr>
</tbody>
</table>

**Important:** The destination register(s) is encoded differently!
Practical attack: Secure Boot

http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.prd29-genc-009492c/ch05s02s01.html
Practical attack: Secure Boot

http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.prd29-genc-009492c/ch05s02s01.html
Boot time attack - Possible approach

1) Destination must be known for the pointer value
2) Original contents in flash must be modified
3) Fault is injected while the pointers are copied
4) Target is compromised when the pointer is loaded into PC
Boot time attack - Possible approach

1) Destination must be known for the pointer value
2) Original contents in flash must be modified
3) Fault is injected while the pointers are copied
4) Target is compromised when the pointer is loaded into PC
Boot time attack - Possible approach

1) Destination must be known for the pointer value
2) Original contents in flash must be modified
3) Fault is injected while the pointers are copied
4) Target is compromised when the pointer is loaded into PC
Boot time attack - Possible approach

1) Destination must be known for the pointer value
2) Original contents in flash must be modified
3) Fault is injected while the pointers are copied
4) Target is compromised when the pointer is loaded into PC
Boot time attack - Possible approach

1) Destination must be known for the pointer value
2) Original contents in flash must be modified
3) Fault is injected while the pointers are copied
4) Target is compromised when the pointer is loaded into PC
Boot time attack - Possible approach

1) Destination must be known for the pointer value
2) Original contents in flash must be modified
3) Fault is injected while the pointers are copied
4) Target is compromised when the pointer is loaded into PC
Boot time attack - Possible approach

1) Destination must be known for the pointer value
2) Original contents in flash must be modified
3) Fault is injected while the pointers are copied
4) Target is compromised when the pointer is loaded into PC
Simulation - LDR

Test code

```c
void print_string (void) { printf("success"); }
void main(void) {
    unsigned int buffer = { &print_string, ... };
    asm volatile ("ldr r0, &buffer;"
                  "ldr r3, [r0];" // target instruction
    );
}
```

Results

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Opcode</th>
</tr>
</thead>
<tbody>
<tr>
<td>ldr r3, [r0]</td>
<td>00000000001100001001000011100101</td>
</tr>
<tr>
<td>ldr pc, [r0]</td>
<td>0000000011110000100100011100101</td>
</tr>
<tr>
<td>ldrle pc, [r0]</td>
<td>000000001111000010010001110101</td>
</tr>
<tr>
<td>ldr pc, [r0, #4]</td>
<td>0000010011110000100100011100101</td>
</tr>
<tr>
<td>ldrne pc, [r0], #8</td>
<td>00001000111100001001000000010100</td>
</tr>
</tbody>
</table>
Simulation - LDR

Test code

```c
1 void print_string (void) { printf("success"); }
2 void main(void) {
3     unsigned int buffer = { &print_string, ... };
4     asm volatile ( 
5         "ldr r0, &buffer;"
6         "ldr r3, [r0];" // target instruction
7     );
8 }
```

Results

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>ldr r3, [r0]</td>
<td>00000000001100001001000011100101</td>
</tr>
<tr>
<td>ldr pc, [r0]</td>
<td>000000001111000010010000110010101</td>
</tr>
<tr>
<td>ldrle pc, [r0]</td>
<td>000000001111000010010000110010101</td>
</tr>
<tr>
<td>ldr pc, [r0, #4]</td>
<td>00000100111100001001000011100101</td>
</tr>
<tr>
<td>ldrne pc, [r0], #8</td>
<td>00001000111100001001000011110100</td>
</tr>
</tbody>
</table>
Test code

```c
void print_string (void) { printf("success"); }
void main(void) {
    unsigned int buffer = { &print_string, ... }
    asm volatile ("ldr r0, &buffer;"
                  "ldr r3, [r0];" // target instruction
    )
}
```

Results

```
lr r3, [r0] 0000000001100001001000011100101
ldr pc, [r0] 00000000111100001001000011100101
ldrle pc, [r0] 00000000111100001001000011100101
ldr pc, [r0, #4] 00000100111100001001000011100101
ldrne pc, [r0], #8 00001000111100001001000000010100
```
Simulation - LDR

Test code

```c
void print_string (void) { printf("success"); }  
void main(void) {
    unsigned int buffer = { &print_string, ... }  
    asm volatile (  
        "ldr r0, &buffer;"
        "ldr r3, [r0];"   // target instruction
    )
}
```

Results

```
ldr  r3, [r0] 00000000001100001001000011100101
ldr  pc, [r0] 0000000111100001001000011100101
ldrle pc, [r0] 00000001111000010010000111010101
ldr  pc, [r0, #4] 0000010011110000101000011100101
ldrne pc, [r0], #8 00001000111100001001000000101000
```
Simulation - LDR

Test code

```c
void print_string (void) { printf("success"); }
void main(void) {
    unsigned int buffer = { &print_string, ... }
    asm volatile ("ldr r0, &buffer;"
                  "ldr r3, [r0];" // target instruction
    )
}
```

Results

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>ldr r3, [r0]</td>
<td>00000000001100001001000011100101</td>
</tr>
<tr>
<td>ldr pc, [r0]</td>
<td>00000000111100001001000011100101</td>
</tr>
<tr>
<td>ldrle pc, [r0]</td>
<td>00000000111100001001000011100101</td>
</tr>
<tr>
<td>ldr pc, [r0, #4]</td>
<td>00000100111110000100100011100101</td>
</tr>
<tr>
<td>ldrne pc, [r0], #8</td>
<td>00001000111100001001000000010100</td>
</tr>
</tbody>
</table>
Simulation - LDMIA

Test code

```c
void print_string (void) { printf("success"); }
void main(void) {
    unsigned int buffer = { &print_string, ... }
    asm volatile ("
        "ldr r0, &buffer;"
        "ldmia r0!, {r4-r7};" /* target instruction */
    )
}
```

Results

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ldmia r0!,{r4-r7}</code></td>
<td>11110000000000001011000011101000</td>
</tr>
<tr>
<td><code>ldmia r0!,{r4-r7,pc}</code></td>
<td>11110000100000001011000011101000</td>
</tr>
<tr>
<td><code>ldmle r0!,{r4-r7, pc}</code></td>
<td>11110000100000001011000011011000</td>
</tr>
<tr>
<td><code>ldmia r0!,{r0,r1,r6,r7,pc}</code></td>
<td>11000011100000001011000011101000</td>
</tr>
<tr>
<td><code>ldmibne r0!,{r0-r3,r8-r14,pc}</code></td>
<td>000011111111111011000000011001</td>
</tr>
</tbody>
</table>
Simulation - LDMIA

**Test code**

```c
1  void print_string (void) { printf("success"); }
2  void main(void) {
3      unsigned int buffer = { &print_string, ... }  
4      asm volatile (
5          "ldr r0, &buffer;"
6          "ldmia r0!, {r4-r7};" // target instruction
7          )
8  }
```

**Results**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>code</th>
</tr>
</thead>
<tbody>
<tr>
<td>ldmia r0!,{r4-r7}</td>
<td>1111000000000001011000011101000</td>
</tr>
<tr>
<td>ldmia r0!,{r4-r7,pc}</td>
<td>1111000010000001011000011101000</td>
</tr>
<tr>
<td>ldmle r0!,{r4-r7, pc}</td>
<td>1111000010000001011000011011000</td>
</tr>
<tr>
<td>ldmia r0!,{r0,r1,r6,r7,pc}</td>
<td>1100001110000001011000011101000</td>
</tr>
<tr>
<td>ldmibne r0!,{r0-r3,r8-r14,pc}</td>
<td>000011111111111101100000011001</td>
</tr>
</tbody>
</table>


Simulation - LDMIA

Test code

```c
void print_string (void) { printf("success"); }
void main(void) {
    unsigned int buffer = { &print_string, ... }
    asm volatile ( "ldr r0, &buffer;
                   ldmia r0!, {r4-r7}＄" // target instruction
    )
}
```

Results

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Signature</th>
<th>Binary Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>ldmia r0!,{r4-r7}</td>
<td></td>
<td>111100000000000001011000011101000</td>
</tr>
<tr>
<td>ldmia r0!,{r4-r7,pc}</td>
<td></td>
<td>11110000100000001011000011101000</td>
</tr>
<tr>
<td>ldmle r0!,{r4-r7, pc}</td>
<td></td>
<td>11110000100000001011000011010000</td>
</tr>
<tr>
<td>ldmia r0!,{r0,r1,r6,r7,pc}</td>
<td></td>
<td>11000011000000001011000011101000</td>
</tr>
<tr>
<td>ldmibne r0!,{r0-r3,r8-r14,pc}</td>
<td></td>
<td>0000111111111110101100000011001</td>
</tr>
</tbody>
</table>
Simulation - LDMIA

Test code

```c
void print_string (void) { printf("success"); }
void main(void) {
    unsigned int buffer = { &print_string, ... };
    asm volatile ("
        ldr r0, &buffer;
        ldmia r0!, {r4-r7};" // target instruction
    )
}
```

Results

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Binary Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>ldmia r0!,{r4-r7}</td>
<td>11110000000000001011000011101000</td>
</tr>
<tr>
<td>ldmia r0!,{r4-r7,pc}</td>
<td>11110000100000001011000011101000</td>
</tr>
<tr>
<td>ldmle r0!,{r4-r7, pc}</td>
<td>11110000100000001011000001101100</td>
</tr>
<tr>
<td>ldmia r0!,{r0,r1,r6,r7,pc}</td>
<td>11000011100000001011000011101000</td>
</tr>
<tr>
<td>ldmibne r0!,{r0-r3,r8-r14,pc}</td>
<td>00001111111111110110000001101001</td>
</tr>
</tbody>
</table>
Simulation - LDMIA

Test code

```c
1     void print_string ( void) { printf("success"); }
2     void main(void) {
3         unsigned int buffer = { &print_string, ... } 
4         asm volatile ( 
5             "ldr r0, &buffer;"
6             "ldmia r0!, {r4-r7};"     // target instruction
7         )
8       }
```

Results

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Machine Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>ldmia r0!,{r4-r7}</td>
<td>11110000000000001011000011101000</td>
</tr>
<tr>
<td>ldmia r0!,{r4-r7,pc}</td>
<td>11110000100000001011000011101000</td>
</tr>
<tr>
<td>ldmle r0!,{r4-r7, pc}</td>
<td>11110000100000001011000011010000</td>
</tr>
<tr>
<td>ldmia r0!,{r0,r1,r6,r7,pc}</td>
<td>11000011100000001011000011101000</td>
</tr>
<tr>
<td>ldmi bne r0!,{r0-r3,r8-r14,pc}</td>
<td>000011111111111101100000011001</td>
</tr>
</tbody>
</table>
Experimentation - Target modification

- Power cut
- Removal of capacitors
- Reset
- Trigger
Experimentation - Target modification

- Power cut
- Removal of capacitors
- Reset
- Trigger
Experimentation - Target modification

- Power cut
- Removal of capacitors
  - Reset
  - Trigger
Experimentation - Target modification

- Power cut
- Removal of capacitors
- Reset
- Trigger
Experimentation - Target modification

- Power cut
- Removal of capacitors
- Reset
- Trigger
Experimentation - Test setup

- USB
- Serial
- Trigger
- Reset
- VCC
- TARGET
void main(void) {
    volatile unsigned int counter = 0;
    set_trigger(1);
    asm volatile ("add r0, r0, #1;" //
        <repeat x1000>     // GLITCH HERE
        "add r0, r0, #1;" //
    );
    set_trigger(0);
    printf("%08x\n", counter);
}
void main(void) {
    volatile unsigned int counter = 0;
    set_trigger(1);
    asm volatile ("add r0, r0, #1;" /* GLITCH HERE */
                  <repeat x1000> /* GLITCH HERE */
                  "add r0, r0, #1;" /* GLITCH HERE */
    );
    set_trigger(0);
    printf("%08x\n", counter);
}
### Experimentation - Characterization

```c
void main(void) {
    volatile unsigned int counter = 0;
    set_trigger(1);
    asm volatile(
        "add r0, r0, #1;" //
        "add r0, r0, #1;" // GLITCH HERE
    );
    set_trigger(0);
    printf("%08x\n", counter);
}
```

*Output 1: 00001000*
*Output 2: 00000ffff*
*Output 3: *

---

*Public*
Experimentation - Characterization

The graph illustrates the relationship between glitch length (in ns) and glitch voltage. The data points are color-coded as follows:

- Green crosses: Expected
- Yellow crosses: Mute
- Red diamonds: Success

The graph shows a trend where longer glitch lengths correspond to lower glitch voltages.
void print_string (void) { printf("success"); }

unsigned int buffer[8] = { &print_string, ...}

void main(void) {
    set_trigger(1);
    asm volatile (
        "ldr r1, =buffer;"
        "ldr r0, [r1];" //
        <repeat x1000> // GLITCH HERE
        "ldr r0, [r1]" //
    );
    set_trigger(0);
    printf("no!");
}

Output 1: "success"
Output 2: "no!"
Output 3: ""
Experimentation - LDR - 10k

Glitch length (ns)

Glitch voltage

+++ Expected
XXX Mute
∗∗∗ Success
```c
void print_string (void) { printf("success"); }

unsigned int buffer[8] = { &print_string, ...}

void main(void) {
    set_trigger(1);
    asm volatile ("ldr r1, =buffer;"
                    "ldmia r0!, r4-r7;" //
                    <repeat x1000>   // GLITCH HERE
                    "ldmia r0!, r4-r7" //
                     );
    set_trigger(0);
    printf("no!");
}
```

Output 1: "success"
Output 2: "no!"
Output 3: ""

Experimentation - LDMIA - 10k

- Glitch length (ns)
- Glitch voltage

Expected, Mute, Success

Public
Countermeasures

- Dedicated hardware countermeasures
  - Fault injection detectors/sensors
  - Integrity checks (e.g. instruction parity)

- Dedicated software countermeasures
  - Deflect (e.g. random delays)
  - Detect (e.g. double check)
  - React (e.g. reset)

- Software exploitation mitigations
  - Only enable execution from memory when needed
  - Randomize copy destination

You can lower the probability but you cannot rule it out!
Countermeasures

- **Dedicated hardware countermeasures**
  - Fault injection detectors/sensors
  - Integrity checks (e.g. instruction parity)

- **Dedicated software countermeasures**
  - Deflect (e.g. random delays)
  - Detect (e.g. double check)
  - React (e.g. reset)

- **Software exploitation mitigations**
  - Only enable execution from memory when needed
  - Randomize copy destination

*You can lower the probability but you cannot rule it out!*
Countermeasures

- Dedicated hardware countermeasures
  - Fault injection detectors/sensors
    - Integrity checks (e.g. instruction parity)

- Dedicated software countermeasures
  - Deflect (e.g. random delays)
  - Detect (e.g. double check)
  - React (e.g. reset)

- Software exploitation mitigations
  - Only enable execution from memory when needed
  - Randomize copy destination

You can lower the probability but you cannot rule it out!
Countermeasures

- Dedicated hardware countermeasures
  - Fault injection detectors/sensors
  - Integrity checks (e.g. instruction parity)

- Dedicated software countermeasures
  - Deflect (e.g. random delays)
  - Detect (e.g. double check)
  - React (e.g. reset)

- Software exploitation mitigations
  - Only enable execution from memory when needed
  - Randomize copy destination

You can lower the probability but you cannot rule it out!
Countermeasures

- Dedicated hardware countermeasures
  - Fault injection detectors/sensors
  - Integrity checks (e.g. instruction parity)

- Dedicated software countermeasures
  - Deflect (e.g. random delays)
  - Detect (e.g. double check)
  - React (e.g. reset)

- Software exploitation mitigations
  - Only enable execution from memory when needed
  - Randomize copy destination

You can lower the probability but you cannot rule it out!
Countermeasures

- Dedicated hardware countermeasures
  - Fault injection detectors/sensors
  - Integrity checks (e.g. instruction parity)

- Dedicated software countermeasures
  - Deflect (e.g. random delays)
    - Detect (e.g. double check)
    - React (e.g. reset)

- Software exploitation mitigations
  - Only enable execution from memory when needed
  - Randomize copy destination

You can lower the probability but you cannot rule it out!
Countermeasures

- Dedicated hardware countermeasures
  - Fault injection detectors/sensors
  - Integrity checks (e.g. instruction parity)

- Dedicated software countermeasures
  - Deflect (e.g. random delays)
  - Detect (e.g. double check)
  - React (e.g. reset)

- Software exploitation mitigations
  - Only enable execution from memory when needed
  - Randomize copy destination

You can lower the probability but you cannot rule it out!
Countermeasures

- Dedicated hardware countermeasures
  - Fault injection detectors/sensors
  - Integrity checks (e.g. instruction parity)

- Dedicated software countermeasures
  - Deflect (e.g. random delays)
  - Detect (e.g. double check)
  - React (e.g. reset)

- Software exploitation mitigations
  - Only enable execution from memory when needed
  - Randomize copy destination

You can lower the probability but you cannot rule it out!
Countermeasures

- Dedicated hardware countermeasures
  - Fault injection detectors/sensors
  - Integrity checks (e.g. instruction parity)

- Dedicated software countermeasures
  - Deflect (e.g. random delays)
  - Detect (e.g. double check)
  - React (e.g. reset)

- Software exploitation mitigations
  - Only enable execution from memory when needed
  - Randomize copy destination

You can lower the probability but you cannot rule it out!
Countermeasures

- Dedicated hardware countermeasures
  - Fault injection detectors/sensors
  - Integrity checks (e.g. instruction parity)

- Dedicated software countermeasures
  - Deflect (e.g. random delays)
  - Detect (e.g. double check)
  - React (e.g. reset)

- Software exploitation mitigations
  - Only enable execution from memory when needed
  - Randomize copy destination

You can lower the probability but you cannot rule it out!
Countermeasures

- Dedicated hardware countermeasures
  - Fault injection detectors/sensors
  - Integrity checks (e.g. instruction parity)

- Dedicated software countermeasures
  - Deflect (e.g. random delays)
  - Detect (e.g. double check)
  - React (e.g. reset)

- Software exploitation mitigations
  - Only enable execution from memory when needed
  - Randomize copy destination

You can lower the probability but you cannot rule it out!
Countermeasures

- Dedicated hardware countermeasures
  - Fault injection detectors/sensors
  - Integrity checks (e.g. instruction parity)

- Dedicated software countermeasures
  - Deflect (e.g. random delays)
  - Detect (e.g. double check)
  - React (e.g. reset)

- Software exploitation mitigations
  - Only enable execution from memory when needed
  - Randomize copy destination

*You can lower the probability but you cannot rule it out!*
Conclusion

- The target is vulnerable to voltage FI
- The PC register on ARM is controllable using FI
  - Combining fault injection and software exploitation is effective
- Success rate is different for `ldr` and `ldmia`
  - The instruction encoding matters
- Software FI countermeasures may not be effective
  - Software exploitation mitigations may complicate attack
- Other instructions and code constructions may be vulnerable
- Other architectures may be vulnerable using specific code constructions
Conclusion

- The target is vulnerable to voltage FI
- The PC register on ARM is controllable using FI
  - Combining fault injection and software exploitation is effective
- Success rate is different for \textit{ldr} and \textit{ldmia}
  - The instruction encoding matters
- Software FI countermeasures may not be effective
  - Software exploitation mitigations may complicate attack
- Other instructions and code constructions may be vulnerable
- Other architectures may be vulnerable using specific code constructions
Conclusion

- The target is vulnerable to voltage FI
- The PC register on ARM is controllable using FI
  - Combining fault injection and software exploitation is effective
- Success rate is different for `ldr` and `ldmia`
  - The instruction encoding matters
- Software FI countermeasures may not be effective
  - Software exploitation mitigations may complicate attack
- Other instructions and code constructions may be vulnerable
- Other architectures may be vulnerable using specific code constructions
Conclusion

- The target is vulnerable to voltage FI
- The PC register on ARM is controllable using FI
  - Combining fault injection and software exploitation is effective
- Success rate is different for ldr and ldmia
  - The instruction encoding matters
- Software FI countermeasures may not be effective
  - Software exploitation mitigations may complicate attack
- Other instructions and code constructions may be vulnerable
- Other architectures may be vulnerable using specific code constructions
Conclusion

- The target is vulnerable to voltage FI

- The PC register on ARM is controllable using FI
  - Combining fault injection and software exploitation is effective

- Success rate is different for *ldr* and *ldmia*
  - The instruction encoding matters

- Software FI countermeasures may not be effective
  - Software exploitation mitigations may complicate attack

- Other instructions and code constructions may be vulnerable

- Other architectures may be vulnerable using specific code constructions
Conclusion

- The target is vulnerable to voltage FI
- The PC register on ARM is controllable using FI
  - Combining fault injection and software exploitation is effective
- Success rate is different for *ldr* and *ldmia*
  - The instruction encoding matters
- Software FI countermeasures may not be effective
  - Software exploitation mitigations may complicate attack
- Other instructions and code constructions may be vulnerable
- Other architectures may be vulnerable using specific code constructions
Conclusion

- The target is vulnerable to voltage FI
- The PC register on ARM is controllable using FI
  - Combining fault injection and software exploitation is effective
- Success rate is different for \textit{ldr} and \textit{ldmia}
  - The instruction encoding matters
- Software FI countermeasures may not be effective
  - Software exploitation mitigations may complicate attack
- Other instructions and code constructions may be vulnerable
- Other architectures may be vulnerable using specific code constructions
Challenge your security

Contact:
Niek Timmers
Senior Security Analyst
timmers@riscure.com

We are hiring!
info@riscure.com
Challenge your security

Contact:
Niek Timmers
Senior Security Analyst
timmers@riscure.com

We are hiring!
inforequest@riscure.com