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

riscure

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

Public

Fault injection techniques

riscure

clock

voltage

electromagnetic

laser

Fault injection fault model

Instruction corruption

```
MOV R0, R1         11100001101000000000000000000000
MOV R0, R2         11100001101000000000000000000010
```

```
MOV R0, R1         11100001101000000000000000000001
STR R7, [R7, #16]  11100101100010010111000000010000
```

Instruction skipping

```
MOV R0, R1         11100001101000000000000000000001
MOV R1, R1         11100001101000000001000000000001
```

```
MOV R0, R1         11100001101000000000000000000001
MOV R6, R6         11100001101000000110000000000110
```

Public

Fault injection fault model

riscure

Instruction corruption

```
MOV R0, R1         11100001101000000000000000000000
MOV R0, R2         11100001101000000000000000000010
```

```
MOV R0, R1         11100001101000000000000000000001
STR R7, [R7, #16]  11100101100010010111000000010000
```

Instruction skipping

```
MOV R0, R1         11100001101000000000000000000001
MOV R1, R1         11100001101000000001000000000001
```

```
MOV R0, R1         11100001101000000000000000000001
MOV R6, R6         11100001101000000110000000000110
```

Public

Fault injection fault model

riscure

Instruction corruption

```
MOV R0, R1          11100001101000000000000000000000
MOV R0, R2          11100001101000000000000000000010
```

```
MOV R0, R1          11100001101000000000000000000001
STR R7, [R7, #16]   11100101100010010111000000010000
```

Instruction skipping

```
MOV R0, R1          11100001101000000000000000000001
MOV R1, R1          11100001101000000001000000000001
```

```
MOV R0, R1          11100001101000000000000000000001
MOV R6, R6          11100001101000000110000000000110
```

Public

Fault injection fault model

riscure

Instruction corruption

```
MOV R0, R1          11100001101000000000000000000000
MOV R0, R2          11100001101000000000000000000010
```

```
MOV R0, R1          11100001101000000000000000000001
STR R7, [R7, #16]   11100101100010010101110000000100000
```

Instruction skipping

```
MOV R0, R1          11100001101000000000000000000001
MOV R1, R1          11100001101000000001000000000001
```

```
MOV R0, R1          11100001101000000000000000000001
MOV R6, R6          11100001101000000110000000000110
```

Public

Fault injection fault model

riscure

Instruction corruption

```
MOV R0, R1         11100001101000000000000000000000
MOV R0, R2         11100001101000000000000000000010
```

```
MOV R0, R1         11100001101000000000000000000001
STR R7, [R7, #16]  11100101100010010111000000010000
```

Instruction skipping

```
MOV R0, R1         11100001101000000000000000000001
MOV R1, R1         11100001101000000001000000000001
```

```
MOV R0, R1         11100001101000000000000000000001
MOV R6, R6         11100001101000000110000000000110
```

Public

Fault injection fault model

riscure

Instruction corruption

```
MOV R0, R1          11100001101000000000000000000000
MOV R0, R2          11100001101000000000000000000010
```

```
MOV R0, R1          11100001101000000000000000000001
STR R7, [R7, #16]  11100101100010010111000000010000
```

Instruction skipping

```
MOV R0, R1          11100001101000000000000000000001
MOV R1, R1          11100001101000000001000000000001
```

```
MOV R0, R1          11100001101000000000000000000001
MOV R6, R6          11100001101000000110000000000110
```

Public

Fault injection fault model

riscure

Instruction corruption

```
MOV R0, R1         11100001101000000000000000000000
MOV R0, R2         11100001101000000000000000000010
```

```
MOV R0, R1         11100001101000000000000000000001
STR R7, [R7, #16]  11100101100010010111000000010000
```

Instruction skipping

```
MOV R0, R1         11100001101000000000000000000001
MOV R1, R1         11100001101000000001000000000001
```

```
MOV R0, R1         11100001101000000000000000000001
MOV R6, R6         11100001101000000110000000000110
```

Public

Why ARM?

riscure

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

Why ARM?

riscure

- ARM is everywhere

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

Why ARM?

riscure

- ARM is everywhere

Application	Chip Function	2015 Device Shipments	Chip Shipments	ARM Chips	Market Share
Mobile Computing *	Apps Processors	1,800	1,800	1,600	>85%
	Connectivity and Control		11,000	4,000	37%
Consumer Electronics **	Apps Processors	3,600	1,000	700	70%
	Connectivity and Control		8,000	3,000	40%
Enterprise Infrastructure	Servers	300	22	>0	<1%
	Networking - Infrastructure		140	20	15%
	Networking - Home and Office		700	200	30%
Automotive	Apps Processors	90	68	65	>95%
	Control		2,700	200	7%
Embedded Intelligence	Apps Processors		500	350	70%
	Connectivity		600	300	50%
	Control		20,000	4,400	22%
Total (in millions)			46,500	14,800	32%

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

Why ARM?

riscure

- ARM is everywhere

Application	Chip Function	2015 Device Shipments	Chip Shipments	ARM Chips	Market Share
Mobile Computing *	Apps Processors Connectivity and Control	1,800	1,800 11,000	1,600 4,000	>85% 37%
Consumer Electronics **	Apps Processors Connectivity and Control	3,600	1,000 8,000	700 3,000	70% 40%
Enterprise Infrastructure	Servers Networking - Infrastructure Networking - Home and Office	300	22 140 700	>0 20 200	<1% 15% 30%
Automotive	Apps Processors Control	90	68 2,700	65 200	>95% 7%
Embedded Intelligence	Apps Processors Connectivity Control		500 600 20,000	350 300 4,400	70% 50% 22%
Total (in millions)			46,500	14,800	32%

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

Why ARM?

riscure

- ARM is everywhere

Application	Chip Function	2015 Device Shipments	2015 Chip Shipments	2015 ARM Chips	2015 Market Share
Mobile Computing *	Apps Processors	1,800	1,800	1,600	>85%
	Connectivity and Control		11,000	4,000	37%
Consumer Electronics **	Apps Processors	3,600	1,000	700	70%
	Connectivity and Control		8,000	3,000	40%
Enterprise Infrastructure	Servers	300	22	>0	<1%
	Networking - Infrastructure		140	20	15%
	Networking - Home and Office		700	200	30%
Automotive	Apps Processors	90	68	65	>95%
	Control		2,700	200	7%
Embedded Intelligence	Apps Processors		500	350	70%
	Connectivity		600	300	50%
	Control		20,000	4,400	22%
Total (in millions)			46,500	14,800	32%

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

All systems copy data from A to B!

Single word copy using LDR / STR

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

Multi-word copy using LDMIA / STMIA

```
1    MultiWorldCopy:
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

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

Multi-word copy using LDMIA / STMIA

```
1       MultiWorldCopy:
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

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

Multi-word copy using LDMIA / STMIA

```
1    MultiWorldCopy:
2        LDMIA r1!, {r3 - r10}
3        STMIA r0!, {r3 - r10}
4        SUBS r2, r2, #32
5        BGE MultiWorldCopy
```

Public

All systems copy data from A to B!

Single word copy using LDR / STR

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

Multi-word copy using LDMIA / STMIA

```
1    MultiWorldCopy:
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

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

Multi-word copy using LDMIA / STMIA

```
1       MultiWorldCopy:
2          LDMIA r1!, {r3 - r10}
3          STMIA r0!, {r3 - r10}
4          SUBS r2, r2, #32
5          BGE MultiWorldCopy
```

Why are copy operations interesting?

riscure

- They operate on attacker controlled data
- They are executed multiple times consecutively
- They are typically not protected

Public

Why are copy operations interesting?

riscure

- They operate on attacker controlled data
- They are executed multiple times consecutively
- They are typically not protected

Why are copy operations interesting?

riscure

- They operate on attacker controlled data
- They are executed multiple times consecutively
- They are typically not protected

Why are copy operations interesting?

riscure

- 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        11100100100100011111000000000100
```

Controlling PC using LDMIA

```
LDMIA r1!,{r3-r10}      11101000101100010000011111111000
```

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

Important: *The destination register(s) is encoded differently!*

Public

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}   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}     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}   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}   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}   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}   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

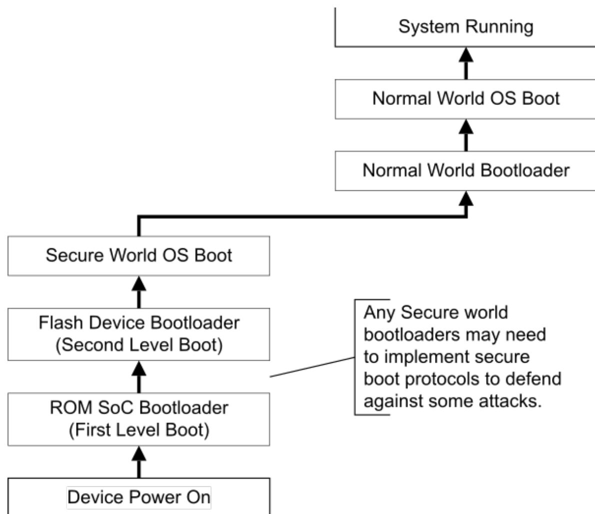
```
LDMIA r1!,{r3-r10}      11101000101100010000011111111000
```

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

Important: *The destination register(s) is encoded differently!*

Practical attack: Secure Boot

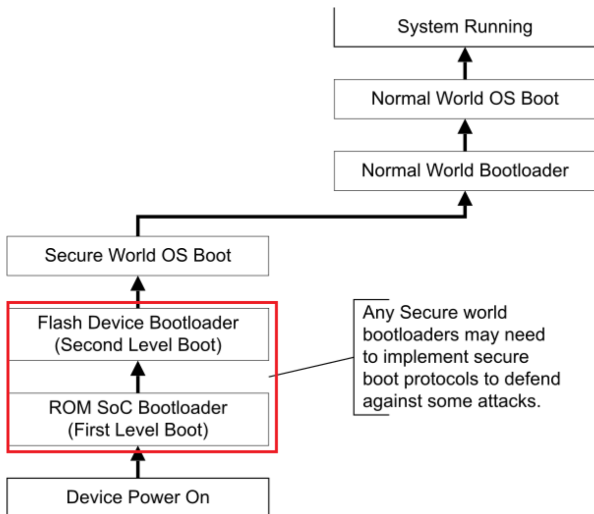
riscure



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

Practical attack: Secure Boot

riscure

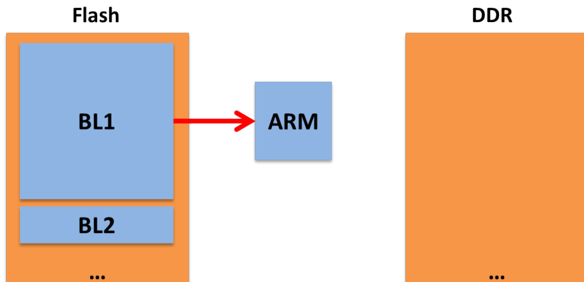


http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.prd29-genc-009492c/ch05s02s01.html

Boot time attack - Possible approach

riscure

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

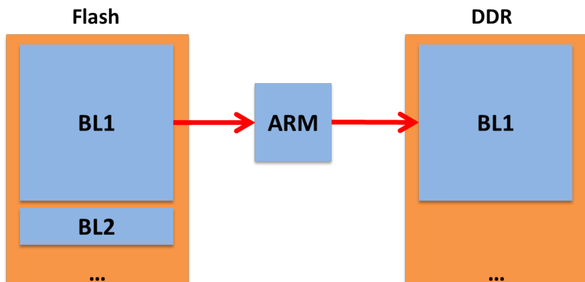


4) Target is compromised when the pointer is loaded into PC

Boot time attack - Possible approach

riscure

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

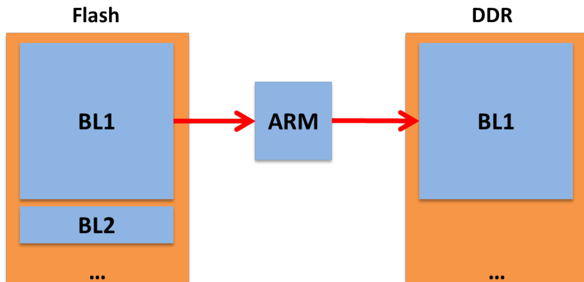


4) Target is compromised when the pointer is loaded into PC

Boot time attack - Possible approach

riscure

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



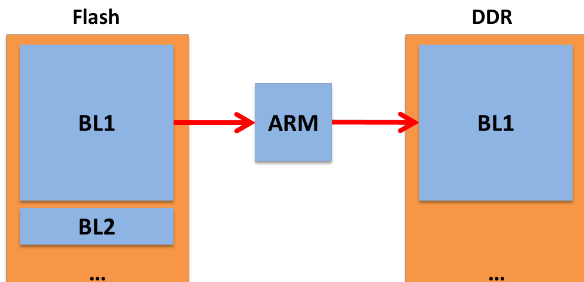
4) Target is compromised when the pointer is loaded into PC

Public

Boot time attack - Possible approach

riscure

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

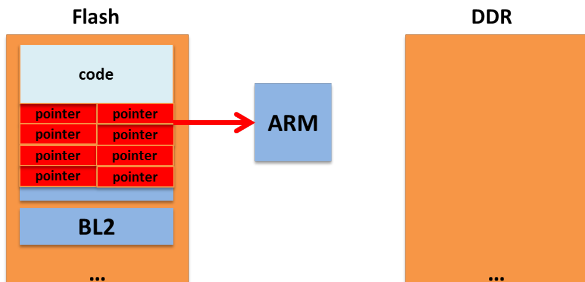


4) Target is compromised when the pointer is loaded into PC

Boost time attack - Possible approach

riscure

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

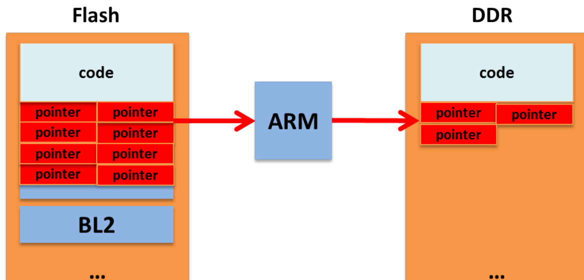


4) Target is compromised when the pointer is loaded into PC

Boot time attack - Possible approach

riscure

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

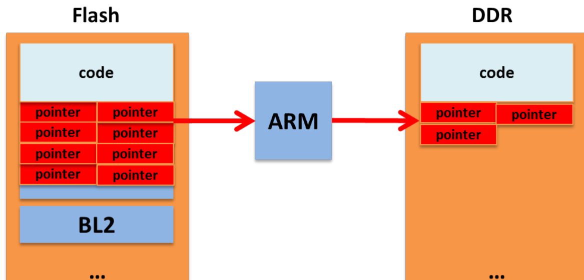


4) Target is compromised when the pointer is loaded into PC

Boot time attack - Possible approach

riscure

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

```
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

```
ldr   r3, [r0]        00000000001100001001000011100101
ldr   pc, [r0]        00000000111100001001000011100101
ldrle pc, [r0]        00000000111100001001000011010101
ldr   pc, [r0, #4]    00000100111100001001000011100101
ldrne pc, [r0], #8    00001000111100001001000000010100
```

Public

Simulation - LDR

Test code

```
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

```
ldr   r3, [r0]        00000000001100001001000011100101
ldr   pc, [r0]        00000000111100001001000011100101
ldrle pc, [r0]        00000000111100001001000011010101
ldr   pc, [r0, #4]    00000100111100001001000011100101
ldrne pc, [r0], #8    00001000111100001001000000010100
```

Public

Simulation - LDR

Test code

```
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

```
ldr   r3, [r0]        00000000001100001001000011100101
ldr   pc, [r0]        00000000111100001001000011100101
ldrle pc, [r0]        00000000111100001001000011010101
ldr   pc, [r0, #4]    00000100111100001001000011100101
ldrne pc, [r0], #8    00001000111100001001000000010100
```

Public

Simulation - LDR

Test code

```
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

```
ldr   r3, [r0]        00000000001100001001000011100101
ldr   pc, [r0]        00000000111100001001000011100101
ldrle pc, [r0]        00000000111100001001000011010101
ldr   pc, [r0, #4]    00000100111100001001000011100101
ldrne pc, [r0], #8    00001000111100001001000000010100
```

Public

Simulation - LDR

Test code

```
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

```
ldr    r3, [r0]        00000000001100001001000011100101
ldr    pc, [r0]        00000000111100001001000011100101
ldrle  pc, [r0]        00000000111100001001000011010101
ldr    pc, [r0, #4]    00000100111100001001000011100101
ldrne  pc, [r0], #8    00001000111100001001000000010100
```

Public

Simulation - LDMIA

Test code

```
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

```
ldmia   r0!,{r4-r7}             11110000000000001011000011101000
ldmia   r0!,{r4-r7,pc}          11110000100000001011000011101000
ldmle   r0!,{r4-r7, pc}         11110000100000001011000011011000
ldmia   r0!,{r0,r1,r6,r7,pc}    11000011100000001011000011101000
ldmibne r0!,{r0-r3,r8-r14,pc}   00001111111111111011000000011001
```

Public

Simulation - LDMIA

Test code

```
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

```
ldmia   r0!,{r4-r7}           11110000000000001011000011101000
ldmia   r0!,{r4-r7,pc}        11110000100000001011000011101000
ldmle   r0!,{r4-r7, pc}       11110000100000001011000011011000
ldmia   r0!,{r0,r1,r6,r7,pc}  11000011100000001011000011101000
ldmibne r0!,{r0-r3,r8-r14,pc} 00001111111111111011000000011001
```

Simulation - LDMIA

Test code

```
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

```
ldmia   r0!,{r4-r7}               11110000000000001011000011101000
ldmia   r0!,{r4-r7,pc}            11110000100000001011000011101000
ldmle   r0!,{r4-r7, pc}           11110000100000001011000011011000
ldmia   r0!,{r0,r1,r6,r7,pc}      11000011100000001011000011101000
ldmibne r0!,{r0-r3,r8-r14,pc}     00001111111111111011000000011001
```

Simulation - LDMIA

Test code

```
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

```
ldmia   r0!,{r4-r7}             11110000000000001011000011101000
ldmia   r0!,{r4-r7,pc}          11110000100000001011000011101000
ldmle   r0!,{r4-r7, pc}         11110000100000001011000011011000
ldmia   r0!,{r0,r1,r6,r7,pc}    11000011100000001011000011101000
ldmibne r0!,{r0-r3,r8-r14,pc}   00001111111111111011000000011001
```

Simulation - LDMIA

Test code

```
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   }
```

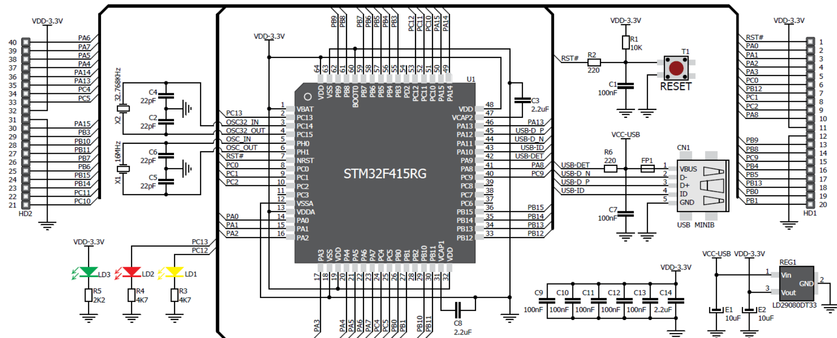
Results

```
ldmia   r0!,{r4-r7}            11110000000000001011000011101000
ldmia   r0!,{r4-r7,pc}         11110000100000001011000011101000
ldmle   r0!,{r4-r7, pc}        11110000100000001011000011011000
ldmia   r0!,{r0,r1,r6,r7,pc}   11000011100000001011000011101000
ldmibne r0!,{r0-r3,r8-r14,pc}  00001111111111111011000000011001
```

Experimentation - Target modification

riscure

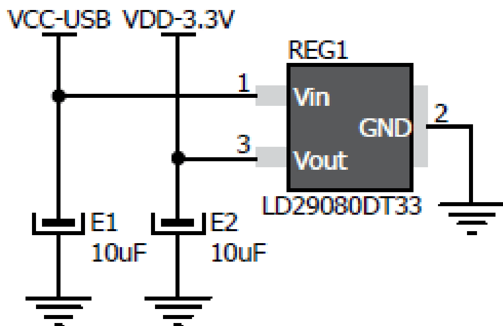
- Power cut
- Removal of capacitors
- Reset
- Trigger



Experimentation - Target modification

riscure

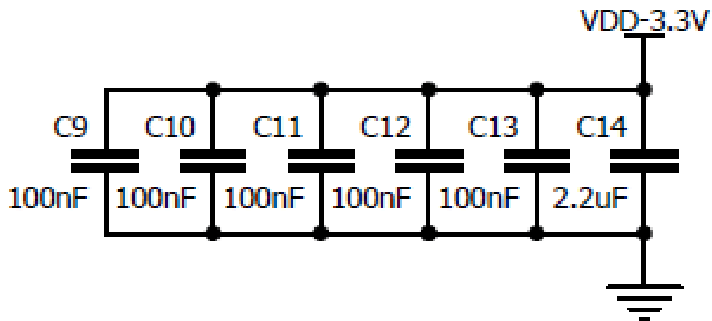
- Power cut
- Removal of capacitors
- Reset
- Trigger



Experimentation - Target modification

riscure

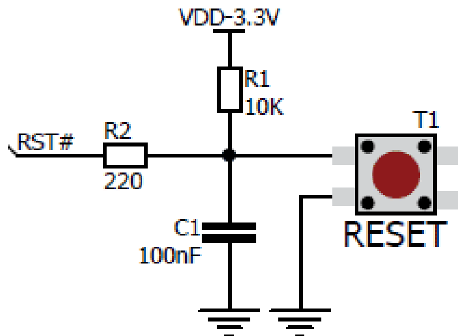
- Power cut
- Removal of capacitors
- Reset
- Trigger



Experimentation - Target modification

riscure

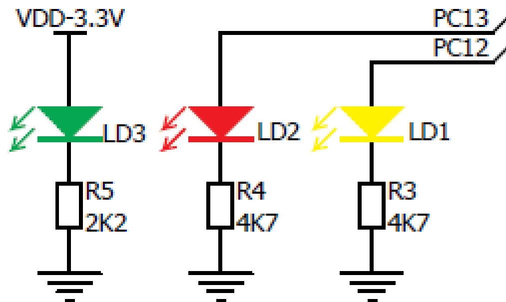
- Power cut
- Removal of capacitors
- Reset
- Trigger



Experimentation - Target modification

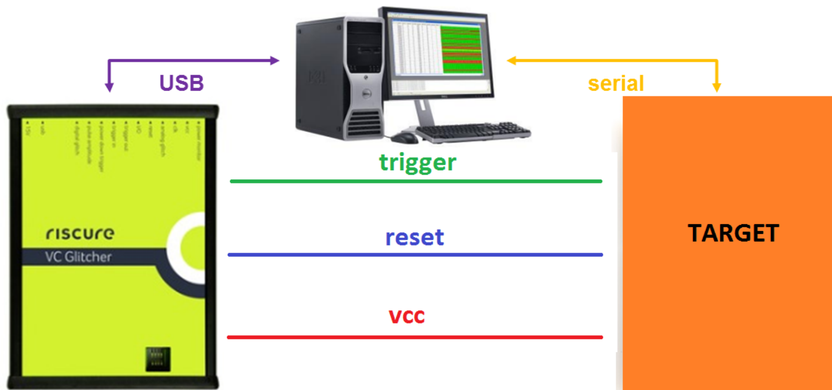
riscure

- Power cut
- Removal of capacitors
- Reset
- Trigger



Experimentation - Test setup

riscure



Public

Experimentation - Characterization

```
1   void main(void) {
2     volatile unsigned int counter = 0;
3     set_trigger(1);
4     asm volatile (
5       "add r0, r0, #1;" //
6         <repeat x1000>  // GLITCH HERE
7       "add r0, r0, #1;" //
8     );
9     set_trigger(0);
10    printf("%08x\n", counter);
11  }
```

```
Output 1: "00001000"
Output 2: "00000fff"
Output 3: ""
```

Experimentation - Characterization

```
1   void main(void) {
2     volatile unsigned int counter = 0;
3     set_trigger(1);
4     asm volatile (
5       "add r0, r0, #1;" //
6         <repeat x1000>  // GLITCH HERE
7       "add r0, r0, #1;" //
8     );
9     set_trigger(0);
10    printf("%08x\n", counter);
11  }
```

```
Output 1: "00001000"
Output 2: "00000fff"
Output 3: ""
```

Experimentation - Characterization

riscure

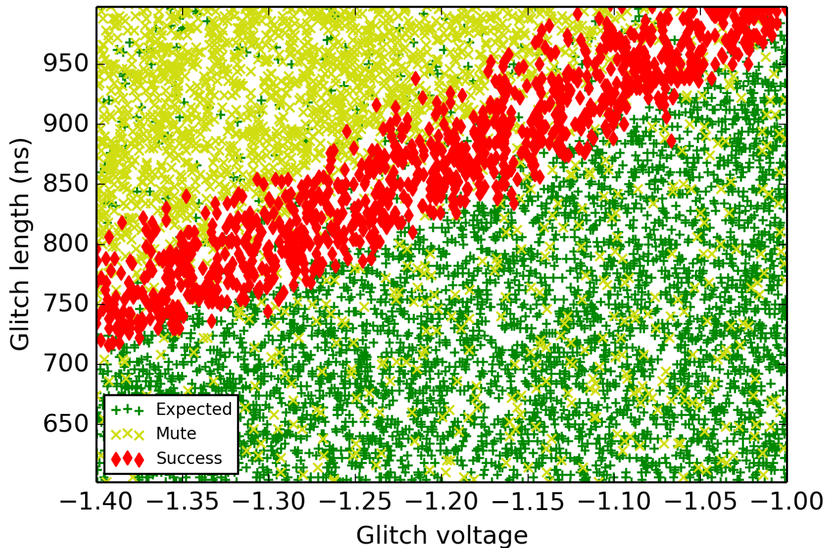
```
1   void main(void) {
2     volatile unsigned int counter = 0;
3     set_trigger(1);
4     asm volatile (
5       "add r0, r0, #1;" //
6         <repeat x1000>  // GLITCH HERE
7       "add r0, r0, #1;" //
8     );
9     set_trigger(0);
10    printf("%08x\n", counter);
11  }
```

```
Output 1: "00001000"
Output 2: "00000fff"
Output 3: ""
```

Public

Experimentation - Characterization

riscure



Public

Experimentation - Test application (LDR)

riscure

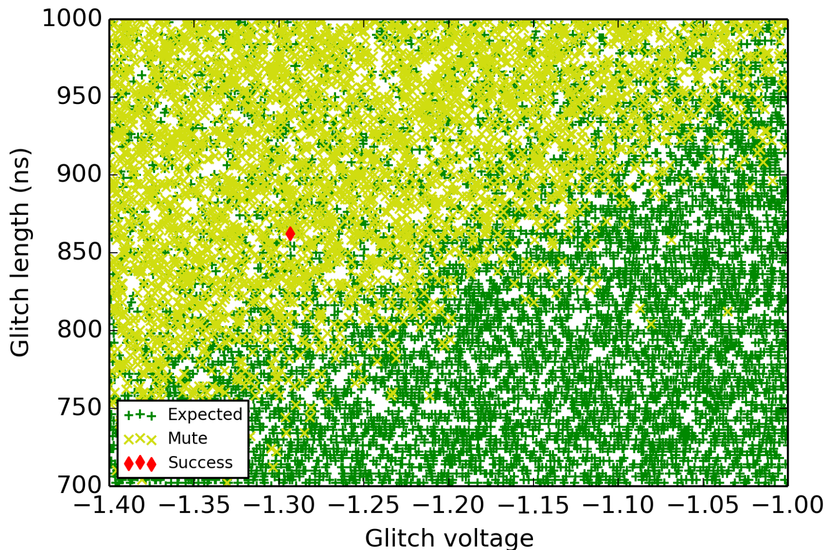
```
1   void print_string (void) { printf("success"); }
2   unsigned int buffer[8] = { &print_string, ...}
3
4   void main(void) {
5     set_trigger(1);
6     asm volatile (
7       "ldr r1, =buffer;"
8       "ldr r0, [r1];"       //
9         <repeat x1000>      // GLITCH HERE
10      "ldr r0, [r1]"        //
11    );
12    set_trigger(0);
13    printf("no!");
14  }
```

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

Public

Experimentation - LDR - 10k

riscure



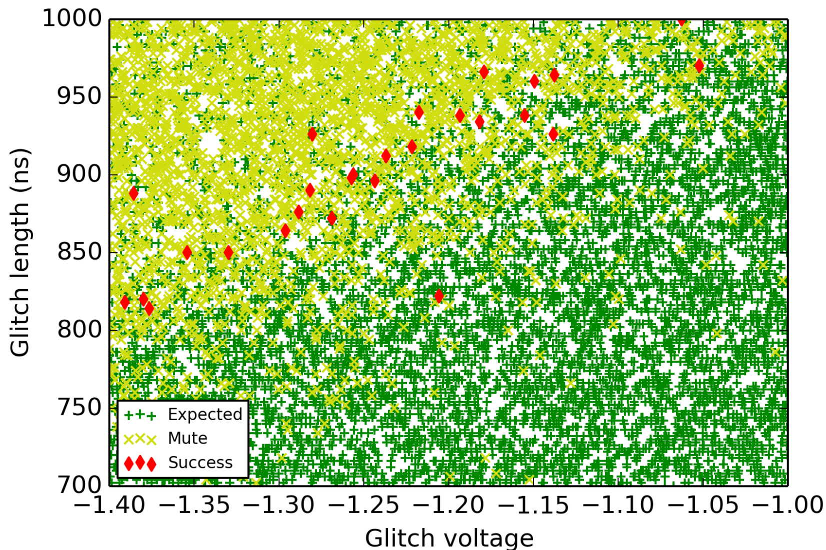
Experimentation - Test application (LDMIA)

```
1   void print_string (void) { printf("success"); }
2   unsigned int buffer[8] = { &print_string, ...}
3
4   void main(void) {
5     set_trigger(1);
6     asm volatile (
7       "ldr r1, =buffer;"
8       "ldmia r0!, r4-r7;"    //
9         <repeat x1000>       // GLITCH HERE
10      "ldmia r0!, r4-r7"     //
11    );
12    set_trigger(0);
13    printf("no!");
14  }
```

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

Public

Experimentation - LDMIA - 10k



riscure

Public

Countermeasures

riscure

- 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

riscure

- 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

riscure

- 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

riscure

- 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

riscure

- 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

riscure

- 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

riscure

- 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

riscure

- 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!

Public

Countermeasures

riscure

- 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

riscure

- 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

riscure

- 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

riscure

- 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 *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

Public

Conclusion

riscure

- 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

Public

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

riscure

Challenge your security

Contact:

Niek Timmers

Senior Security Analyst

timmers@riscure.com

We are hiring!

inforequest@riscure.com

riscure

Challenge your security

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

We are hiring!

inforequest@riscure.com