Bypassing Secure Boot using Fault Injection

Niek Timmers
timmers@riscure.com
(@tieknimmers)

Albert Spruyt
spruyt@riscure.com

November 4, 2016
What are the contents of this talk?

**Keywords** – fault injection, secure boot, bypasses, mitigations, practicalities, best practices, demo(s) ...
Who are we?

Albert & Niek

- (Senior) Security Analysts at Riscure
- Security testing of different products and technologies

Riscure

- Services (Security Test Lab)
  - Hardware / Software / Crypto
  - Embedded systems / Smart cards
- Tools
  - Side channel analysis (passive)
  - Fault injection (active)
- Offices
  - Delft, The Netherlands / San Francisco, USA

Combining services and tools for fun and profit!
Who are we?

Albert & Niek
- (Senior) Security Analysts at Riscure
- Security testing of different products and technologies

Riscure
- Services (Security Test Lab)
  - Hardware / Software / Crypto
  - Embedded systems / Smart cards
- Tools
  - Side channel analysis (passive)
  - Fault injection (active)
- Offices
  - Delft, The Netherlands / San Francisco, USA

Combining services and tools for fun and profit!
Who are we?

Albert & Niek

- (Senior) Security Analysts at Riscure
- Security testing of different products and technologies

Riscure

- Services (Security Test Lab)
  - Hardware / Software / Crypto
  - Embedded systems / Smart cards
- Tools
  - Side channel analysis (passive)
  - Fault injection (active)
- Offices
  - Delft, The Netherlands / San Francisco, USA

Combining services and tools for fun and profit!
Fault Injection – A definition...

"Introducing faults in a target to alter its intended behavior."

... 
if( key_is_correct ) <-- Glitch here! 
{
  open_door();
}
else 
{
  keep_door_closed();
}
...

How can we introduce these faults?
Fault Injection – A definition...

“Introducing faults in a target to alter its intended behavior.”

...  
if( key_is_correct ) <-- Glitch here!  
{  
   open_door();  
}  
else  
{  
   keep_door_closed();  
}  
...  

How can we introduce these faults?
Fault Injection – A definition...

"Introducing faults in a target to alter its intended behavior."

... if( key_is_correct ) <-- Glitch here! 
{
    open_door();
}
else
{
    keep_door_closed();
}
...

How can we introduce these faults?
Fault Injection – A definition...

"Introducing faults in a target to alter its intended behavior."

... 

if( key_is_correct ) <-- Glitch here! 
{
  open_door();
}
else 
{
  keep_door_closed();
}
... 

How can we introduce these faults?
Fault injection techniques¹

Remark
- All techniques introduce faults externally

¹ The Sorcerers Apprentice Guide to Fault Attacks. – Bar-El et al., 2004
Fault injection techniques\(^1\)

Remark

- All techniques introduce faults externally

---

\(^1\) The Sorcerers Apprentice Guide to Fault Attacks. – Bar-El et al., 2004
Voltage fault injection

- Pull the voltage down at the right moment
- Not 'too soft'; Not 'too hard'

Source: http://www.limited-entropy.com/fault-injection-techniques/
Fault models

Faults that affect hardware

- Registers
- Buses

Faults that affect hardware that does software

- Instruction corruption
- Data corruption

The true fault model is hard to predict or prove!

---

2 Fault Model Analysis of Laser-Induced Faults in SRAM Memory Cells – Roscian et. al., 2015
3 High Precision Fault Injections on the Instruction Cache of ARMv7-M Architectures – Riviere et al., 2015
4 Formal verification of a software countermeasure against instruction skip attacks – Moro et. al., 2014
Fault models

Faults that affect hardware

- Registers
- Buses

Faults that affect hardware that does software

- Instruction corruption
- Data corruption

The true fault model is hard to predict or prove!

---

2 Fault Model Analysis of Laser-Induced Faults in SRAM Memory Cells – Roscian et. al., 2015
3 High Precision Fault Injections on the Instruction Cache of ARMv7-M Architectures – Riviere et al., 2015
4 Formal verification of a software countermeasure against instruction skip attacks – Moro et. al., 2014
Fault models

Faults that affect hardware

- Registers
- Buses

Faults that affect hardware that does software

- Instruction corruption
- Data corruption

*The true fault model is hard to predict or prove!*

---

2 Fault Model Analysis of Laser-Induced Faults in SRAM Memory Cells – Roscian et. al., 2015
3 High Precision Fault Injections on the Instruction Cache of ARMv7-M Architectures – Riviere et al., 2015
4 Formal verification of a software countermeasure against instruction skip attacks – Moro et. al., 2014
Fault models

Faults that affect hardware

• Registers
• Buses

Faults that affect hardware that does software

• Instruction corruption
• Data corruption

The true fault model is hard to predict or prove!

---

2 Fault Model Analysis of Laser-Induced Faults in SRAM Memory Cells – Roscian et. al., 2015
3 High Precision Fault Injections on the Instruction Cache of ARMv7-M Architectures – Riviere et al., 2015
4 Formal verification of a software countermeasure against instruction skip attacks – Moro et. al., 2014
Fault Models – "Our" choice ...

When presented with code: instruction corruption.

Simple (MIPS)

```
addi $t1, $t1, 8  00100001001010010000000000001000
addi $t1, $t1, 0  00100001001010010000000000000000
```

Complex (ARM)

```
ldr w1, [sp, #0x8] 10111001010000000000101111100001
str w7, [sp, #0x20] 101110010000000000100011111100111
```

Remarks

- Limited control over which bit(s) will be corrupted
- May or may not be the true fault model
- Other fault model behavior covered
Fault Models – “Our” choice ...

When presented with code: instruction corruption.

**Simple (MIPS)**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Binary Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>addi $t1, $t1, 8</code></td>
<td>00100001001010010000000000001000</td>
</tr>
<tr>
<td><code>addi $t1, $t1, 0</code></td>
<td>00100001001010010000000000000000</td>
</tr>
</tbody>
</table>

**Complex (ARM)**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Binary Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ldr w1, [sp, #0x8]</code></td>
<td>1011100101000000000010111100001</td>
</tr>
<tr>
<td><code>str w7, [sp, #0x20]</code></td>
<td>10111001000000000000100111100111</td>
</tr>
</tbody>
</table>

**Remarks**

- Limited control over which bit(s) will be corrupted
- May or may not be the true fault model
- Other fault model behavior covered
Fault Models – ”Our” choice ...

When presented with code: instruction corruption.

Simple (MIPS)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Machine Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>addi $t1, $t1, 8</td>
<td>00100001001010010000000000001000</td>
</tr>
<tr>
<td>addi $t1, $t1, 0</td>
<td>00100001001010010000000000000000</td>
</tr>
</tbody>
</table>

Complex (ARM)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Machine Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>ldr w1, [sp, #0x8]</td>
<td>1011100101000000000000001011111000001</td>
</tr>
<tr>
<td>str w7, [sp, #0x20]</td>
<td>101110010000000000000111100111</td>
</tr>
</tbody>
</table>

Remarks

- Limited control over which bit(s) will be corrupted
- May or may not be the true fault model
- Other fault model behavior covered
Fault Models – "Our" choice ...

When presented with code: instruction corruption.

**Simple (MIPS)**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>addi $t1, $t1, 8</td>
<td>001000010010100100000000000001000</td>
</tr>
<tr>
<td>addi $t1, $t1, 0</td>
<td>0010000100101001000000000000000000</td>
</tr>
</tbody>
</table>

**Complex (ARM)**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>ldr w1, [sp, #0x8]</td>
<td>101110010100000000000101111100001</td>
</tr>
<tr>
<td>str w7, [sp, #0x20]</td>
<td>1011100100000000001000111110000011</td>
</tr>
</tbody>
</table>

**Remarks**

- Limited control over which bit(s) will be corrupted
- May or may not be the true fault model
- Other fault model behavior covered
Fault Models – "Our" choice ...

When presented with code: instruction corruption.

Simple (MIPS)

```plaintext
addi $t1, $t1, 8  00100001001010010000000000001000
addi $t1, $t1, 0  00100001001010010000000000000000
```

Complex (ARM)

```plaintext
ldr w1, [sp, #0x8]  10111001010000000000101111000001
str w7, [sp, #0x20] 101110010000000000110011111000111
```

Remarks

- Limited control over which bit(s) will be corrupted
- May or may not be the true fault model
- Other fault model behavior covered
Fault Models – “Our” choice ...

When presented with code: instruction corruption.

Simple (MIPS)

```
addi $t1, $t1, 8  00100001001010010000000000001000
addi $t1, $t1, 0  00100001001010010000000000000000
```

Complex (ARM)

```
ldr w1, [sp, #0x8]  1011100101000000000010111100001
str w7, [sp, #0x20] 101110010000000000100111110011
```

Remarks

- Limited control over which bit(s) will be corrupted
- May or may not be the true fault model
- Other fault model behavior covered
Fault Models – "Our" choice ...

When presented with code: *instruction corruption*.

**Simple (MIPS)**

```
addi $t1, $t1, 8 00100001001010010000000000001000
addi $t1, $t1, 0 00100001001010010000000000000000
```

**Complex (ARM)**

```
ldr w1, [sp, #0x8] 1011100101000000000010111100001
str w7, [sp, #0x20] 1011100100000000001001111100111
```

**Remarks**

- Limited control over which bit(s) will be corrupted
- May or may not be the true fault model
- Other fault model behavior covered
Why is there Secure Boot?

Remarks

- Integrity and confidentiality of flash contents are not assured!
- A mechanism is required for this assurance: secure boot
Why is there Secure Boot?

Remarks

- Integrity and confidentiality of flash contents are not assured!
- A mechanism is required for this assurance: secure boot
Why is there Secure Boot?

Remarks

- Integrity and confidentiality of flash contents are not assured!
- A mechanism is required for this assurance: secure boot
Why is there Secure Boot?

Remarks

- Integrity and confidentiality of flash contents are not assured!
- A mechanism is required for this assurance: secure boot
Why is there Secure Boot?

Remarks

- Integrity and confidentiality of flash contents are not assured!
- A mechanism is required for this assurance: **secure boot**
Secure Boot – Generic Design

- Assures **integrity** (and **confidentiality**) of flash contents
- The **chain of trust** is similar to PKI\(^5\) found in browsers
- One **root of trust** composed of immutable code and key

\(^5\)Public Key Infrastructure
Secure Boot – Generic Design

- Assures **integrity** (and **confidentiality**) of flash contents
- The **chain of trust** is similar to PKI\(^5\) found in browsers
- One **root of trust** composed of immutable code and key

---

\(^5\) Public Key Infrastructure
Secure Boot – Generic Design

- Assures **integrity** (and **confidentiality**) of flash contents
- The **chain of trust** is similar to PKI\(^5\) found in browsers
- One **root of trust** composed of immutable code and key

\(^{5}\) Public Key Infrastructure
Secure Boot – Generic Design

- Assures **integrity** (and **confidentiality**) of flash contents
- The **chain of trust** is similar to PKI\(^5\) found in browsers
- One **root of trust** composed of immutable code and key

\(^5\) Public Key Infrastructure
Secure Boot – In reality ...

**Normal World**
- To Hypervisor / Linux Kernel
  - BL33
    - Non Trusted Firmware to load the Non Secure OS.
      - (e.g.: U-Boot, EDK2)

**Secure World**
- BL32
  - Secure EL1 payload
  - Trusted OS kernel
- BL2
  - Trusted Boot Firmware
  - Trusted boot board
- BL1
  - AP Boot ROM
  - Trusted boot board
  - 1st level Boot Loader loads 2nd level image
  - 2nd level Boot Loader loads all 3rd level images

**Key**
- EL3 Execution
- Secure EL1 Execution
- Normal EL2/EL1 Execution

**Glossary**
- EDK2 – EFI Development Kit 2
- EL – Exception Level
- PSCI – Power State Control Interface
- BL – Boot Loader
- SMC – Secure Monitor Call

**Source:** http://community.arm.com/docs/DOC-9306
Secure Boot – In reality...

**Normal World**

- To Hypervisor / Linux Kernel
- **BL33**
  - Non-Trusted Firmware: Load the Non Secure OS. (e.g. U-Boot, EDK2)

**Secure World**

- **BL32**
  - Secure EL1 payload
  - Trusted OS kernel
- **BL2**
  - Trusted Boot Firmware
  - Trusted boot board
- **BL1**
  - AP Boot ROM
  - Trusted boot board

1. 1st level Boot Loader loads 2nd level image
2. BL2 Trust Boot Firmware
3. BL32 Secure EL1 payload
4. BL31 EL3 Runtime Firmware
5. BL33 Non-Trusted Firmware: Load the Non Secure OS. (e.g. U-Boot, EDK2)

**Key**

- EL3 Execution
- Secure EL1 Execution
- Normal EL2/EL1 Execution

**Glossary**

- EDK2 – EFI Development Kit 2
- EL – Exception Level
- PSCI – Power State Control Interface
- BL – Boot Loader
- SMC – Secure Monitor Call

**Source:** http://community.arm.com/docs/DOC-9306
Why use a hardware attack?

"Logical issues exist in secure boot implementations!!?"

Bootloader vulnerabilities

- S5L8920 (iPhone)\(^6\)
- Amlogic S905\(^7\)

However

- Small code base results in a small logical attack surface
- Implementations without vulnerabilities likely exist

Other attack(s) must be used when not logically flawed!

---

\(^6\) [https://www.theiphonewiki.com/wiki/0x24000_Segment_Overflow](https://www.theiphonewiki.com/wiki/0x24000_Segment_Overflow)

\(^7\) [http://www.fredericb.info/2016/10/amlogic-s905-soc-bypassing-not-so.html](http://www.fredericb.info/2016/10/amlogic-s905-soc-bypassing-not-so.html)
Why use a hardware attack?

"Logical issues exist in secure boot implementations!!?"

Bootloader vulnerabilities

- S5L8920 (iPhone)\(^6\)
- Amlogic S905\(^7\)

However

- Small code base results in a small logical attack surface
- Implementations without vulnerabilities likely exist

Other attack(s) must be used when not logically flawed!

---

\(^6\) [https://www.theiphonewiki.com/wiki/0x24000_Segment_Overflow](https://www.theiphonewiki.com/wiki/0x24000_Segment_Overflow)

\(^7\) [http://www.fredericb.info/2016/10/amlogic-s905-soc-bypassing-not-so.html](http://www.fredericb.info/2016/10/amlogic-s905-soc-bypassing-not-so.html)
Why use a hardware attack?

"Logical issues exist in secure boot implementations!!?"

Bootloader vulnerabilities

- S5L8920 (iPhone)\(^6\)
- Amlogic S905\(^7\)

However

- Small code base results in a small logical attack surface
- Implementations without vulnerabilities likely exist

Other attack(s) must be used when not logically flawed!

---

\(^6\) [https://www.theiphonewiki.com/wiki/0x24000_Segment_Overflow](https://www.theiphonewiki.com/wiki/0x24000_Segment_Overflow)

\(^7\) [http://www.fredericb.info/2016/10/amlogic-s905-soc-bypassing-not-so.html](http://www.fredericb.info/2016/10/amlogic-s905-soc-bypassing-not-so.html)
Why use a hardware attack?

"Logical issues exist in secure boot implementations!!?"

Bootloader vulnerabilities

- S5L8920 (iPhone)\(^6\)
- Amlogic S905\(^7\)

However

- Small code base results in a small logical attack surface
- Implementations without vulnerabilities likely exist

Other attack(s) must be used when not logically flawed!

---

\(^6\) [https://www.theiphonewiki.com/wiki/0x24000_Segment_Overflow](https://www.theiphonewiki.com/wiki/0x24000_Segment_Overflow)

\(^7\) [http://www.fredericb.info/2016/10/amlogic-s905-soc-bypassing-not-so.html](http://www.fredericb.info/2016/10/amlogic-s905-soc-bypassing-not-so.html)
Why use a hardware attack?

"Logical issues exist in secure boot implementations!!?"

Bootloader vulnerabilities

- S5L8920 (iPhone)\(^6\)
- Amlogic S905\(^7\)

However

- Small code base results in a small logical attack surface
- Implementations without vulnerabilities likely exist

*Other attack(s) must be used when not logically flawed!*

\(^6\) [https://www.theiphonewiki.com/wiki/0x24000_Segment_Overflow](https://www.theiphonewiki.com/wiki/0x24000_Segment_Overflow)

\(^7\) [http://www.fredericb.info/2016/10/amlogic-s905-soc-bypassing-not-so.html](http://www.fredericb.info/2016/10/amlogic-s905-soc-bypassing-not-so.html)
Why use a hardware attack?

"Logical issues exist in secure boot implementations!!?"

Bootloader vulnerabilities

- S5L8920 (iPhone)\(^6\)
- Amlogic S905\(^7\)

However

- Small code base results in a small logical attack surface
- Implementations without vulnerabilities likely exist

Other attack(s) must be used when not logically flawed!

---

\(^6\) https://www.theiphonewiki.com/wiki/0x24000_Segment_Overflow
\(^7\) http://www.fredericb.info/2016/10/amlogic-s905-soc-bypassing-not-so.html
Why use a hardware attack?

"Logical issues exist in secure boot implementations!!?"

Bootloader vulnerabilities

- S5L8920 (iPhone)
- Amlogic S905

However

- Small code base results in a small logical attack surface
- Implementations without vulnerabilities likely exist

Other attack(s) must be used when not logically flawed!

---

6 https://www.theiphonewiki.com/wiki/0x24000_Segment_Overflow
7 http://www.fredericb.info/2016/10/amlogic-s905-soc-bypassing-not-so.html
Why (not) fault injection on secure boot?

Cons
- Invasive
- Physical access
- Expensive

Pros
- No logical vulnerability required
- Typical targets not properly protected

*Especially relevant when assets are not available after boot!*
Why (not) fault injection on secure boot?

**Cons**
- Invasive
- Physical access
- Expensive

**Pros**
- No logical vulnerability required
- Typical targets not properly protected

*Especially relevant when assets are not available after boot!*
Why (not) fault injection on secure boot?

**Cons**
- Invasive
- Physical access
- Expensive

**Pros**
- No logical vulnerability required
- Typical targets not properly protected

*Especially relevant when assets are not available after boot!*
Why (not) fault injection on secure boot?

Cons

• Invasive
• Physical access
• Expensive

Pros

• No logical vulnerability required
• Typical targets not properly protected

Especially relevant when assets are not available after boot!
Typical assets

Secure code
- Boot code (ROM\textsuperscript{8})

Secrets
- Keys (for boot code decryption)

Secure hardware
- Cryptographic engines

\textsuperscript{8} Read Only Memory
Typical assets

Secure code
- Boot code (ROM\textsuperscript{8})

Secrets
- Keys (for boot code decryption)

Secure hardware
- Cryptographic engines

\textsuperscript{8} Read Only Memory
Typical assets

Secure code
- Boot code (ROM\(^8\))

Secrets
- Keys (for boot code decryption)

Secure hardware
- Cryptographic engines

\(^8\) Read Only Memory
Typical assets

Secure code
- Boot code (ROM\textsuperscript{8})

Secrets
- Keys (for boot code decryption)

Secure hardware
- Cryptographic engines

\textsuperscript{8} Read Only Memory
Fault Injection – Intermezzo
Fault Injection – Tooling

Micah posted a very nice video using the ChipWhisperer-Lite

By NewAE Technology Inc.
Fault Injection – Tooling

Micah posted a very nice video using the ChipWhisperer-Lite

By NewAE Technology Inc.
Fault Injection – Setup

Target

- Digilent Zybo (Xilinx Zynq-7010 System-on-Chip)
- ARM Cortex-A9 (AArch32)
Fault Injection – Setup

Target
- Digilent Zybo (Xilinx Zynq-7010 System-on-Chip)
- ARM Cortex-A9 (AArch32)
Fault Injection – Setup
Characterization – Test application

```
asm volatile
(
    ...
    "add r1, r1, #1;"
    "add r1, r1, #1;"
    < repeat >  <-- glitch here
    "add r1, r1, #1;"
    "add r1, r1, #1;"
    ...
)
```

Remarks

- Full control over the target
- Increasing a counter using ADD instructions
- Send counter back using the serial interface

\[^{11}\] Implemented as an U-Boot command
Characterization – Possible responses

Expected: ’too soft’
counter = 00010000

Mute: ’too hard’
counter =

Success: ’$$$’
counter = 00009999
counter = 00010015
counter = 00008687

Remarks
• Glitching ’too hard’ may damage the target permanently
Characterization – Possible responses

Expected: ’too soft’
counter = 00010000

Mute: ’too hard’
counter =

Success: ’$$$’
counter = 00009999
counter = 00010015
counter = 00008687

Remarks

• Glitching ’too hard’ may damage the target permanently
Characterization – Possible responses

Expected: ’too soft’

counter = 00010000

Mute: ’too hard’

counter =

Success: ’$$$

counter = 00009999
counter = 00010015
counter = 00008687

Remarks

- Glitching ’too hard’ may damage the target permanently
Characterization – Possible responses

Expected: 'too soft'

- counter = 00010000

Mute: 'too hard'

- counter =

Success: '$$$'

- counter = 00009999
- counter = 00010015
- counter = 00008687

Remarks

- Glitching 'too hard' may damage the target permanently
Characterization – Possible responses

Expected: ‘too soft’
counter = 00010000

Mute: ‘too hard’

Success: ‘$$$’
counter = 00009999
counter = 00010015
counter = 00008687

Remarks
• Glitching ‘too hard’ may damage the target permanently
DEMO 1
PARAMETER SEARCH

Glitch parameters
- Randomize glitch delay within the attack window
- Randomize the glitch voltage
- Randomize the glitch length
DEMO 1
PARAMETER SEARCH

Glitch parameters
- Randomize glitch delay within the attack window
- Randomize the glitch voltage
- Randomize the glitch length
DEMO 1
PARAMETER SEARCH

Glitch parameters

- Randomize glitch delay within the attack window
- Randomize the glitch voltage
- Randomize the glitch length
Glitch parameters

- Randomize glitch delay within the attack window
- Randomize the glitch voltage
- Randomize the glitch length
That was the introduction ... 

... let's bypass secure boot: The Classics!
That was the introduction ...

... let’s bypass secure boot: The Classics!
Classic Bypass 00: Hash comparison

- Applicable to all secure boot implementations
- Bypass of authentication

```c
if( memcmp( p, hash, hashlen ) != 0 )
    return( MBEDTLS_ERR_RSA_VERIFY_FAILED );

p += hashlen;

if( p != end )
    return( MBEDTLS_ERR_RSA_VERIFY_FAILED );

return( 0 );
```

Source: https://tls.mbed.org/
Classic Bypass 00: Hash comparison

- Applicable to all secure boot implementations
- Bypass of authentication

```c
if( memcmp( p, hash, hashlen ) != 0 )
    return( MBEDTLS_ERR_RSA_VERIFY_FAILED );

p += hashlen;

if( p != end )
    return( MBEDTLS_ERR_RSA_VERIFY_FAILED );

return( 0 );
```

Source: https://tls.mbed.org/
Classic Bypass 00: Hash comparison

- Applicable to all secure boot implementations
- Bypass of authentication

```c
if( memcmp( p, hash, hashlen ) != 0 )
    return( MBEDTLS_ERR_RSA_VERIFY_FAILED );

p += hashlen;

if( p != end )
    return( MBEDTLS_ERR_RSA_VERIFY_FAILED );

return( 0 );
```

Source: https://tls.mbed.org/
Multiple locations bypass the check with a single fault!
Classic Bypass 00: Hash comparison

Multiple locations bypass the check with a single fault!
Classic Bypass 00: Hash comparison

Multiple locations bypass the check with a single fault!
Classic Bypass 00: Hash comparison

Multiple locations bypass the check with a single fault!
Classic Bypass 00: Hash comparison

Multiple locations bypass the check with a single fault!
Classic Bypass 01: Signature check call

/* glitch here */
if(mbedtls_pk_verify(&k, SHA256, h, hs, s, ss)) {
    /* do not boot up the image */
    no_boot();
} else {
    /* boot up the image */
    boot();
}

Remarks

- Bypasses can happen on all levels
- Inside functions, inside the calling functions, etc.
Classic Bypass 01: Signature check call

```c
/* glitch here */
if (mbedtls_pk_verify(&k, SHA256, h, hs, s, ss)) {
    /* do not boot up the image */
    no_boot();
} else {
    /* boot up the image */
    boot();
}
```

Remarks

- Bypasses can happen on all levels
- Inside functions, inside the calling functions, etc.
Classic Bypass 01: Signature check call

/* glitch here */
if (mbedtls_pk_verify(&k, SHA256, h, hs, s, ss)) {
  /* do not boot up the image */
  no_boot();
} else {
  /* boot up the image */
  boot();
}

Remarks

- Bypasses can happen on all levels
- Inside functions, inside the calling functions, etc.
Classic Bypass 02: Infinite loop

- What to do when the signature verification fails?
  - Enter an infinite loop!

```c
/* glitch here */
if(mbedtls_pk_verify(&k, SHA256, h, hs, s, ss)) {
    /* do not boot up the image */
    while(1);
} else {
    /* boot up the image */
    boot();
}
```
Classic Bypass 02: Infinite loop

- What to do when the signature verification fails?
- Enter an infinite loop!

```c
/* glitch here */
if(mbedtls_pk_verify(&k, SHA256, h, hs, s, ss)) {
    /* do not boot up the image */
    while(1);
}
else {
    /* boot up the image */
    boot();
}
```
Classic Bypass 02: Infinite loop

- What to do when the signature verification fails?
  - Enter an infinite loop!

```c
/* glitch here */
if (mbedtls_pk_verify(&k, SHA256, h, hs, s, ss)) {
    /* do not boot up the image */
    while (1);
}
else {
    /* boot up the image */
    boot();
}
```
Classic Bypass 02: Infinite loop

- What to do when the signature verification fails?
- Enter an infinite loop!

```c
/* glitch here */
if(mbedtls_pk_verify(&k, SHA256, h, hs, s, ss)) {
    /* do not boot up the image */
    while(1);
}
else {
    /* boot up the image */
    boot();
}
```
Classic Bypass 02: Infinite loop

Remarks

- Timing is not an issue!
- Classic smart card attack
- Better to reset or wipe keys

Classic Bypass 02: Infinite loop

Remarks

- Timing is not an issue!
- Classic smart card attack
- Better to reset or wipe keys

Classic Bypass 02: Infinite loop

Remarks

- Timing is not an issue!
- Classic smart card attack
- Better to reset or wipe keys

Classic Bypass 02: Infinite loop

Remarks

- Timing is not an issue!
- Classic smart card attack
- Better to reset or wipe keys

Classic Bypass 02: Infinite loop

Remarks

- Timing is not an issue!
- Classic smart card attack \(^{12}\)
- Better to reset or wipe keys

Classic Bypass 03: Secure boot enable

- Secure boot often enabled/disabled based on OTP\textsuperscript{13} bit
- No secure boot during development; secure boot in the field
- Typically just after the CPU comes out of reset

\textsuperscript{13}One-Time-Programmable memory
Fault Injection – Mitigations

Hardware countermeasures

- Detect the glitch or fault

Software countermeasures

- Lower the probability of a successful fault
- Do not address the root cause

You can lower the probability but not rule it out!

---

14 The Sorcerers Apprentice Guide to Fault Attacks – Bar-El et al., 2004
15 The Fault Attack Jungle - A Classification Model to Guide You – Verbauwhede et al., 2011
16 Secure Application Programming in the Presence of Side Channel Attacks – Witteman
Fault Injection – Mitigations

Hardware countermeasures

- Detect the glitch or fault

Software countermeasures

- Lower the probability of a successful fault
- Do not address the root cause

You can lower the probability but not rule it out!

---

14 The Sorcerers Apprentice Guide to Fault Attacks – Bar-El et al., 2004
15 The Fault Attack Jungle - A Classification Model to Guide You – Verbauwhede et al., 2011
16 Secure Application Programming in the Presence of Side Channel Attacks – Witteman
Fault Injection – Mitigations

Hardware countermeasures

• Detect the glitch or fault

Software countermeasures

• Lower the probability of a successful fault
• Do not address the root cause

You can lower the probability but not rule it out!

---

14 The Sorcerers Apprentice Guide to Fault Attacks – Bar-El et al., 2004
15 The Fault Attack Jungle - A Classification Model to Guide You – Verbauwhede et al., 2011
16 Secure Application Programming in the Presence of Side Channel Attacks – Witteman
Fault Injection – Mitigations

Hardware countermeasures

- Detect the glitch or fault

Software countermeasures

- Lower the probability of a successful fault
- Do not address the root cause

You can lower the probability but not rule it out!

---

14 The Sorcerers Apprentice Guide to Fault Attacks – Bar-El et al., 2004
15 The Fault Attack Jungle - A Classification Model to Guide You – Verbauwhede et al., 2011
16 Secure Application Programming in the Presence of Side Channel Attacks – Witteman
Fault Injection – Mitigations

Hardware countermeasures

- Detect the glitch or fault

Software countermeasures

- Lower the probability of a successful fault
- Do not address the root cause

You can *lower the probability* but not rule it out!

---

14 The Sorcerers Apprentice Guide to Fault Attacks – Bar-El et al., 2004
15 The Fault Attack Jungle - A Classification Model to Guide You – Verbauwhede et al., 2011
16 Secure Application Programming in the Presence of Side Channel Attacks – Witteman
Fault Injection – Mitigations

Hardware countermeasures

- Detect the glitch or fault

Software countermeasures

- Lower the probability of a successful fault
- Do not address the root cause

You can *lower the probability* but *not rule it out!*

---

14 The Sorcerers Apprentice Guide to Fault Attacks – Bar-El et al., 2004
15 The Fault Attack Jungle - A Classification Model to Guide You – Verbauwhede et al., 2011
16 Secure Application Programming in the Presence of Side Channel Attacks – Witteman
Compiler optimizations

Why?
- ROM memory size is limited
- Compiler optimizations decrease code size

*Compiler optimizes out intended code!*
Compiler optimizations

Why?

- ROM memory size is limited
- Compiler optimizations decrease code size

*Compiler optimizes out intended code*!
Compiler optimizations

Why?

- ROM memory size is limited
- Compiler optimizations decrease code size

Compiler optimizes out intended code!
Example of a double check

```c
unsigned int compare(char * input, int len)
{
    if (memcmp(password, input, len) == 0) <-- 1st
    {
        if (memcmp(password, input, len) == 0) <-- 2nd
        {
            return TRUE;
        }
    }
    return FALSE;
}
```
Compiler 'optimization' – Double check

Compiled **without** optimizations
Compiler ’optimization’ – Double check

Compiled with optimizations

; int __fastcall compare(void *s2, size_t n)
EXPORT compare
compare
PUSH   {R3,LR}
MOV    R2, R1 ; n
MOV    R1, R0 ; s2
MOV    R0, #aPassword ; s1
BLX    memcmp
ADDS   R0, #0
IT NE
MOVNE  R0, #1
NEG    R0, R0
POP    {R3,PC}
; End of function compare
Compiler ’optimizations’ – Best practices

- Your compiler is smarter than you

- Use ’volatile’ to prevent compiler problems

- Read the output of the compiler!
Compiler ‘optimizations’ – Best practices

- Your compiler is smarter than you
- Use ‘volatile’ to prevent compiler problems
- Read the output of the compiler!
Your compiler is smarter than you

Use ‘volatile’ to prevent compiler problems

Read the output of the compiler!
Compiler ’optimizations’ – Best practices

- Your compiler is smarter than you
- Use ’volatile’ to prevent compiler problems
- Read the output of the compiler!
Example of a double check using 'volatile'

```c
int checkSecureBoot() {
    volatile int * otp_secure_boot = OTP_SECURE_BOOT;

    if ( (*otp_secure_boot >> 7) & 0x1 ) { <-- 1st
        return 0;
    } else {
        if ( (*otp_secure_boot >> 7) & 0x1 ) { <-- 2nd
            return 0;
        } else {
            return 1;
        }
    }
}
```
Compiler ’optimization’ – Pointer setup

Compiled with optimizations

checkSecureBoot
MOV      R3, #0x20204000
LDR     R2, [R3] ; Load from pointer
LSLS    R2, R2, #0x18
ITTTE PL
LDRPL   R0, [R3] ; Second load from pointer
UBFXPL.W R0, R0, #7, #1
EORPL.W R0, R0, #1
MOVMI   R0, #0
BX      LR
Compiler ’optimization’ – Pointer setup

Compiled with optimizations

```assembly
checkSecureBoot
MOV
LDR
LSLS
ITTTE PL
LDRPL
UBFXPL.W
EORPL.W
MOVMI
BX
R3, #0x20204000 ; Load from pointer
R2, R2, #0x18
R0, [R3] ; Second load from pointer
R0, R0, #7, #1
R0, R0, #1
R0, #0
LR
```
Combined Attacks

Those were the classics and their mitigations..

... the attack surface is larger!
Combined Attacks

Those were the classics and their mitigations..

... the attack surface is larger!\(^{17}\)

\(^{17}\) All attacks have been performed successfully on multiple targets!
Combined attack – Copy

- Introducing logical vulnerabilities using fault injection
  - Build your own buffer overflow!
- Easy approach: change `memcpy` the size argument

Before corruption

```c
memcpy(dst, src, 0x1000);
```

After corruption

```c
memcpy(dst, src, 0xCEE5);
```

Remark

- Works when dedicated hardware is used (e.g. DMA\textsuperscript{18} engines)

\textsuperscript{18}Direct Memory Access
Combined attack – Copy

- Introducing logical vulnerabilities using fault injection
  - Build your own buffer overflow!
- Easy approach: change `memcpy` the size argument

Before corruption

```c
memcpy(dst, src, 0x1000);
```

After corruption

```c
memcpy(dst, src, 0xCEE5);
```

Remark

- Works when dedicated hardware is used
  (e.g. DMA\textsuperscript{18} engines)

\textsuperscript{18}Direct Memory Access
**Combined attack – Copy**

- Introducing logical vulnerabilities using fault injection
  - Build your own buffer overflow!
- Easy approach: change `memcpy` the size argument

**Before corruption**

```c
memcpy(dst, src, 0x1000);
```

**After corruption**

```c
memcpy(dst, src, 0xCEE5);
```

**Remark**

- Works when dedicated hardware is used (e.g. DMA\(^\text{18}\) engines)

---

\(^\text{18}\) Direct Memory Access
Combined attack – Copy

- Introducing logical vulnerabilities using fault injection
  - Build your own buffer overflow!
- Easy approach: change `memcpy` the size argument

Before corruption

```c
memcpy(dst, src, 0x1000);
```

After corruption

```c
memcpy(dst, src, 0xCEE5);
```

Remark

- Works when dedicated hardware is used (e.g. DMA\(^\text{18}\) engines)

\(^{18}\) Direct Memory Access
Combined attack – Copy

- Introducing logical vulnerabilities using fault injection
  - Build your own buffer overflow!
- Easy approach: change `memcpy` the size argument

Before corruption

```c
memcpy(dst, src, 0x1000);
```

After corruption

```c
memcpy(dst, src, 0xCEE5);
```

Remark

- Works when dedicated hardware is used (e.g. DMA\textsuperscript{18} engines)

\textsuperscript{18}Direct Memory Access
Combined attack – Copy

Remark

• Targeting the copy function arguments
Combined attack – Copy

- Targetting the copy function arguments
Remark

- Targeting the copy function arguments
Combined attack – Copy

Generic Embedded System - on

```
LDR  R2, [SP, #0x10] ; size
MOV  R1, R4 ; src
MOV  R0, R5 ; dst
BL   memcpy
``` 

Remark
- Targetting the copy function arguments
Combined attack – Copy

Generic Embedded System - on

LDR R2, [SP+4]  
MOV R1, R4  
MOV R0, R5  
BL memcpy

Remark
- Targeting the copy function arguments
Remark

- Targetting the copy function arguments
Combined attack - Controlling PC on ARM

- Exploits an ARM32 characteristic
- PC register is directly accessible by most instructions

**Multi-word copy**

```
LDMIA r1!, {r3 - r10}
STMIA r0!, {r3 - r10}
```

**Controlling PC using LDMIA**

```
LDMIA r1!,{r3-r10} 11101000101100010000011111111000
LDMIA r1!,{r3-r10,PC} 11101000101100011000011111111000
```

- Variations possible on other architectures; code dependent

---

19 Program Counter
20 Controlling PC on ARM using Fault Injection – Timmers et al., 2016
Combined attack - Controlling PC on ARM

- Exploits an ARM32 characteristic
- PC\textsuperscript{19} register is directly accessible by most instructions

**Multi-word copy**

```
LDMIA r1!, {r3 - r10}
STMIA r0!, {r3 - r10}
```

**Controlling PC using LDMIA**

```
LDMIA r1!,{r3-r10} 11101000101100010000011111111000
LDMIA r1!,{r3-r10,PC} 1110100010110001100011111111000
```

- Variations possible on other architectures; code dependent

\textsuperscript{19} Program Counter

\textsuperscript{20} Controlling PC on ARM using Fault Injection – Timmers et al., 2016
Combined attack - Controlling PC on ARM

- Exploits an ARM32 characteristic
- PC\textsuperscript{19} register is directly accessible by most instructions

\textit{Multi-word copy}

\begin{verbatim}
LDMIA r1!, {r3 - r10}
STMIA r0!, {r3 - r10}
\end{verbatim}

\textit{Controlling PC using LDMIA}

\begin{verbatim}
LDMIA r1!,{r3-r10} 11101000101100010000011111111000
LDMIA r1!,{r3-r10,PC} 11101000101100011000011111111000
\end{verbatim}

- Variations possible on other architectures; code dependent

\textsuperscript{19}Program Counter
\textsuperscript{20}Controlling PC on ARM using Fault Injection – Timmers et al., 2016
Combined attack - Controlling PC on ARM

- Exploits an ARM32 characteristic
- PC register is directly accessible by most instructions

Multi-word copy

LDMIA r1!, {r3 - r10}
STMIA r0!, {r3 - r10}

Controlling PC using LDMIA

LDMIA r1!,{r3-r10} 111010001011000100000111111111000
LDMIA r1!,{r3-r10,PC} 111010001011000110000111111111000

- Variations possible on other architectures; code dependent

---

19 Program Counter
20 Controlling PC on ARM using Fault Injection – Timmers et al., 2016
Combined attack - Controlling PC on ARM

Generic Embedded System - on

... LDR R2, [SP, #0x10] ; size
MOV R1, R4 ; src
MOV R0, R5 ; dst
BL memcpy
...

Remark
- Targetting the copy function arguments
Combined attack - Controlling PC on ARM

Remark
- Targetting the copy function arguments
Combined attack - Controlling PC on ARM

Generic Embedded System - on

```
LDR  R2, [SP, #0x10] ; size
MOV  R1, R4 ; src
MOV  R0, R5 ; dst
BL   memcpy
...
```

Remark

- Targetting the copy function arguments
Combined attacks - Wild jungle jump

- Start glitching while/after loading the image but before decryption
- Lots of ‘magic’ pointers around, which point close to the code
- Get them from: stack, register, memory
- The more magic pointers, the higher the probability

Proving the wild jungle jump – Gratchoff, 2015
Combined attacks - Wild jungle jump

- Start glitching while/after loading the image but before decryption
- Lots of 'magic' pointers around, which point close to the code
- Get them from: stack, register, memory
- The more magic pointers, the higher the probability

21 Proving the wild jungle jump – Gratchoff, 2015
Combined attacks - Wild jungle jump

- Start glitching while/after loading the image but before decryption
- Lots of ’magic’ pointers around, which point close to the code
  - Get them from: stack, register, memory
  - The more magic pointers, the higher the probability

---

21 Proving the wild jungle jump – Gratchoff, 2015
Combined attacks - Wild jungle jump

- Start glitching while/after loading the image but before decryption
- Lots of 'magic' pointers around, which point close to the code
- Get them from: stack, register, memory
- The more magic pointers, the higher the probability

---

21 Proving the wild jungle jump – Gratchoff, 2015
• Start glitching while/after loading the image but before decryption

• Lots of 'magic' pointers around, which point close to the code

• Get them from: stack, register, memory

• The more magic pointers, the higher the probability

---

21 Proving the wild jungle jump – Gratchoff, 2015
Combined attack(s) – Summary

- Bypass of both authentication and decryption
- Typically little software exploitation mitigation during boot
- Fault injection mitigations in software may not be effective
Combined attack(s) – Summary

- Bypass of both authentication and decryption
- Typically little software exploitation mitigation during boot
- Fault injection mitigations in software may not be effective
Combined attack(s) – Summary

- Bypass of both authentication and decryption
- Typically little software exploitation mitigation during boot
- Fault injection mitigations in software may not be effective
Combined attack(s) – Summary

- Bypass of both authentication and decryption
- Typically little software exploitation mitigation during boot
- Fault injection mitigations in software may not be effective
There are some practicalities ...

... which we must overcome!
There are some practicalities ...

... which we must overcome!
Secure Boot – Demo Design

Remark

- Stage 2 is invalidated by changing the printed string
- Stage 1 enters an infinite loop when the signature is invalid
Remark

- Stage 2 is invalidated by changing the printed string
- Stage 1 enters an infinite loop when the signature is invalid
Secure Boot – Demo Design

Remark

- Stage 2 is invalidated by changing the printed string
- Stage 1 enters an infinite loop when the signature is invalid
When to glitch?

- Not possible to use a signal originating from target
- Only reference point is power-on reset moment
- Use side-channels to obtain more information
- Compare behavior between valid image and an invalid image
When to glitch?

- Not possible to use a signal originating from target
- Only reference point is power-on reset moment
- Use side-channels to obtain more information
- Compare behavior between valid image and an invalid image
When to glitch?

- Not possible to use a signal originating from target
- Only reference point is power-on reset moment
- Use side-channels to obtain more information
- Compare behavior between valid image and an invalid image

ATTACK HERE!
When to glitch?

- Not possible to use a signal originating from target
- Only reference point is power-on reset moment
- Use side-channels to obtain more information
- Compare behavior between valid image and an invalid image
When to glitch?

- Not possible to use a signal originating from target
- Only reference point is power-on reset moment
- Use side-channels to obtain more information
- Compare behavior between valid image and an invalid image
When to glitch?

- Not possible to use a signal originating from target
- Only reference point is power-on reset moment
- Use side-channels to obtain more information
- Compare behavior between valid image and an invalid image

**NOT POSSIBLE!**
When to glitch?

- Not possible to use a signal originating from target
- Only reference point is power-on reset moment
- Use side-channels to obtain more information
- Compare behavior between valid image and an invalid image

**NOT POSSIBLE!**
Boot profiling – Reset

Valid image

Invalid image

Remark
- No difference between a valid and invalid image
- Attack window spreads across the entire trace (≈400 ms)
Boot profiling – Reset

Valid image

Invalid image

Remark

- No difference between a valid and invalid image
- Attack window spreads across the entire trace (~400 ms)
Boot profiling – Reset

Valid image

Invalid image

Remark
- No difference between a valid and invalid image
- Attack window spreads across the entire trace (~400 ms)
Boot profiling – Flash activity

Valid image

Invalid image

Remarks

- Flash activity 3 not present for the invalid image
- Attack window between flash activity 2 and 3 (~10 ms)
Boot profiling – Flash activity

Valid image

Invalid image

Remarks

- Flash activity 3 not present for the invalid image
- Attack window between flash activity 2 and 3 (~10 ms)
Boot profiling – Flash activity

Valid image

Invalid image

Remarks

- Flash activity 3 not present for the invalid image
- Attack window between flash activity 2 and 3 (~10 ms)
Boot profiling – Power consumption

Remark

- Measuring electromagnetic emissions using a probe

Boot profiling – Power consumption

Remark

• Measuring electromagnetic emissions using a probe

Boot profiling – Power consumption

Remarks

- Significant difference in the electromagnetic emissions
- Attack window reduced significantly (< 1 ms)
- Power profile at black arrow is flat: infinite loop
Remarks

- Significant difference in the electromagnetic emissions
- Attack window reduced significantly (< 1 ms)
- Power profile at black arrow is flat: infinite loop
Remarks

- Significant difference in the electromagnetic emissions
- Attack window reduced significantly (< 1 ms)
- Power profile at black arrow is flat: infinite loop
Boot profiling – Power consumption

Remarks

- Significant difference in the electromagnetic emissions
- Attack window reduced significantly (< 1 ms)
- Power profile at black arrow is flat: infinite loop
Boot profiling – Power consumption

**Remarks**

- Significant difference in the electromagnetic emissions
- Attack window reduced significantly (< 1 ms)
- Power profile at black arrow is flat: infinite loop
Jitter during boot prevents effective timing (~150 µs)
Remark

- Jitter during boot prevents effective timing (~150 µs)
Remark

- Jitter during boot prevents effective timing (~150 µs)
How to minimize jitter during boot?

- Power-on reset is too early
- Use a signal close to the 'glitch moment'

Remark
- Using flash activity 2 as a trigger to minimize jitter
How to minimize jitter during boot?

- Power-on reset is too early
  - Use a signal close to the 'glitch moment'

Remark
- Using flash activity 2 as a trigger to minimize jitter
How to minimize jitter during boot?

- Power-on reset is too early
- Use a signal close to the 'glitch moment'

Remark
- Using flash activity 2 as a trigger to minimize jitter
How to minimize jitter during boot?

- Power-on reset is too early
- Use a signal close to the 'glitch moment'

Remark
- Using flash activity 2 as a trigger to minimize jitter
How to minimize jitter during boot?

- Power-on reset is too early
- Use a signal close to the 'glitch moment'

Remark
- Using flash activity 2 as a trigger to minimize jitter
Remarks

- Jitter minimized using flash activity as a trigger
Glitch Timing – Power consumption

Remarks

- Jitter minimized using flash activity as a trigger
DEMO 2
BYPASSING SECURE BOOT

Glitch parameter search
- Fixed the glitch delay to 300 ms
- Fixed the glitch voltage to -2 V
- Randomize the glitch length
DEMO 2

BYPASSING SECURE BOOT

Glitch parameter search

- Fixed the glitch delay to 300 ms
- Fixed the glitch voltage to -2 V
- Randomize the glitch length
DEMO 2
BYPASSING SECURE BOOT

Glitch parameter search

- Fixed the glitch delay to 300 ms
- Fixed the glitch voltage to -2 V
- Randomize the glitch length
DEMO 2
BYPASSING SECURE BOOT

Glitch parameter search

- Fixed the glitch delay to 300 ms
- Fixed the glitch voltage to -2 V
- Randomize the glitch length
DEMO 2

BYPASSING SECURE BOOT
Secure Boot – Manufacturer Best practices

Minimize attack surface
- Authenticate all code and data
- Limit functionality in ROM code
- Disable memory when not required

Lower the probability
- Implement fault injection countermeasures
- Implement software exploitation mitigations

*Robustness can only be determined using testing!*
Secure Boot – Manufacturer Best practices

Minimize attack surface

- Authenticate all code and data
- Limit functionality in ROM code
- Disable memory when not required

Lower the probability

- Implement fault injection countermeasures
- Implement software exploitation mitigations

Robustness can only be determined using testing!
Secure Boot – Manufacturer Best practices

Minimize attack surface

- Authenticate all code and data
- Limit functionality in ROM code
- Disable memory when not required

Lower the probability

- Implement fault injection countermeasures
- Implement software exploitation mitigations

Robustness can only be determined using testing!
Secure Boot – Manufacturer Best practices

Minimize attack surface

- Authenticate all code and data
- Limit functionality in ROM code
- Disable memory when not required

Lower the probability

- Implement fault injection countermeasures
- Implement software exploitation mitigations

Robustness can only be determined using testing!
Secure Boot – Manufacturer Best practices

**Minimize attack surface**
- Authenticate all code and data
- Limit functionality in ROM code
- Disable memory when not required

**Lower the probability**
- Implement fault injection countermeasures
- Implement software exploitation mitigations

Robustness can only be determined using testing!
Secure Boot – Manufacturer Best practices

Minimize attack surface

- Authenticate all code and data
- Limit functionality in ROM code
- Disable memory when not required

Lower the probability

- Implement fault injection countermeasures
- Implement software exploitation mitigations

Robustness can only be determined using testing!
Secure Boot – Manufacturer Best practices

Minimize attack surface
- Authenticate all code and data
- Limit functionality in ROM code
- Disable memory when not required

Lower the probability
- Implement fault injection countermeasures
- Implement software exploitation mitigations

Robustness can only be determined using testing!
Secure Boot – Manufacturer Best practices

Minimize attack surface

- Authenticate all code and data
- Limit functionality in ROM code
- Disable memory when not required

Lower the probability

- Implement fault injection countermeasures
- Implement software exploitation mitigations

Robustness can only be determined using testing!
Secure Boot – Manufacturer Best practices

Minimize attack surface
- Authenticate all code and data
- Limit functionality in ROM code
- Disable memory when not required

Lower the probability
- Implement fault injection countermeasures
- Implement software exploitation mitigations

Robustness can only be determined using testing!
• Today’s standard technology not resistant to fault attacks

• Implementers of secure boot should address fault risks

• Hardware fault injection countermeasures are needed

• Fault injection testing provides assurance on product security
Conclusion / Sound Bytes

- Today’s standard technology not resistant to fault attacks
  - Implementers of secure boot should address fault risks
  - Hardware fault injection countermeasures are needed
  - Fault injection testing provides assurance on product security
Today’s standard technology not resistant to fault attacks

Implementers of secure boot should address fault risks

Hardware fault injection countermeasures are needed

Fault injection testing provides assurance on product security
Conclusion / Sound Bytes

- Today’s standard technology not resistant to fault attacks
- Implementers of secure boot should address fault risks
- Hardware fault injection countermeasures are needed
- Fault injection testing provides assurance on product security
• Today’s standard technology not resistant to fault attacks

• Implementers of secure boot should address fault risks

• Hardware fault injection countermeasures are needed

• Fault injection testing provides assurance on product security
Challenge your security

Niek Timmers
Senior Security Analyst
timmers@riscure.com (@tieknimmers)

Albert Spruyt
Senior Security Analyst
spruyt@riscure.com

www.riscure.com/careers
inforequest@riscure.com