AVRAND: A Software-Based Defense Against Code Reuse Attacks for AVR Embedded Devices

Sergio Pastrana, Juan Tapiador,
Guillermo Suarez-Tangil, Pedro Peris-Lopez

DIMVA 2016
San Sebastian. 7,8 July 2016

Outline

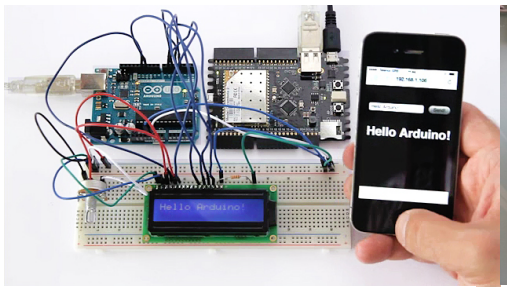
Outline

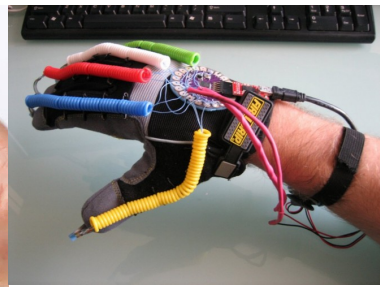
The security of AVR devices has not been properly considered

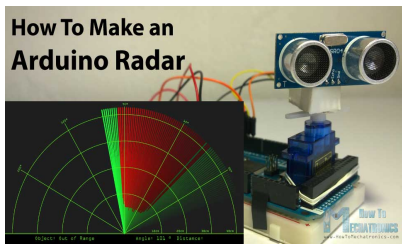
- IoT involves a huge variety of architectures
  - ARM, MIPS, x86, **AVR**...
- Security and safety of these devices is critical
  - Connectivity ("thingbots")
  - Critical scenarios
- Some challenges
  - Resource constrained devices
  - New exploitation vectors

The security of AVR devices has not been properly considered

- IoT involves a huge variety of architectures
  - ARM, MIPS, x86, **AVR**...
- Security and safety of these devices is critical
  - Connectivity ("thingbots")
  - Critical scenarios
- Some challenges
  - Resource constrained devices
  - New exploitation vectors



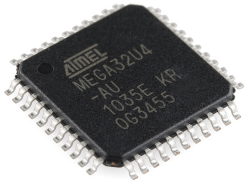




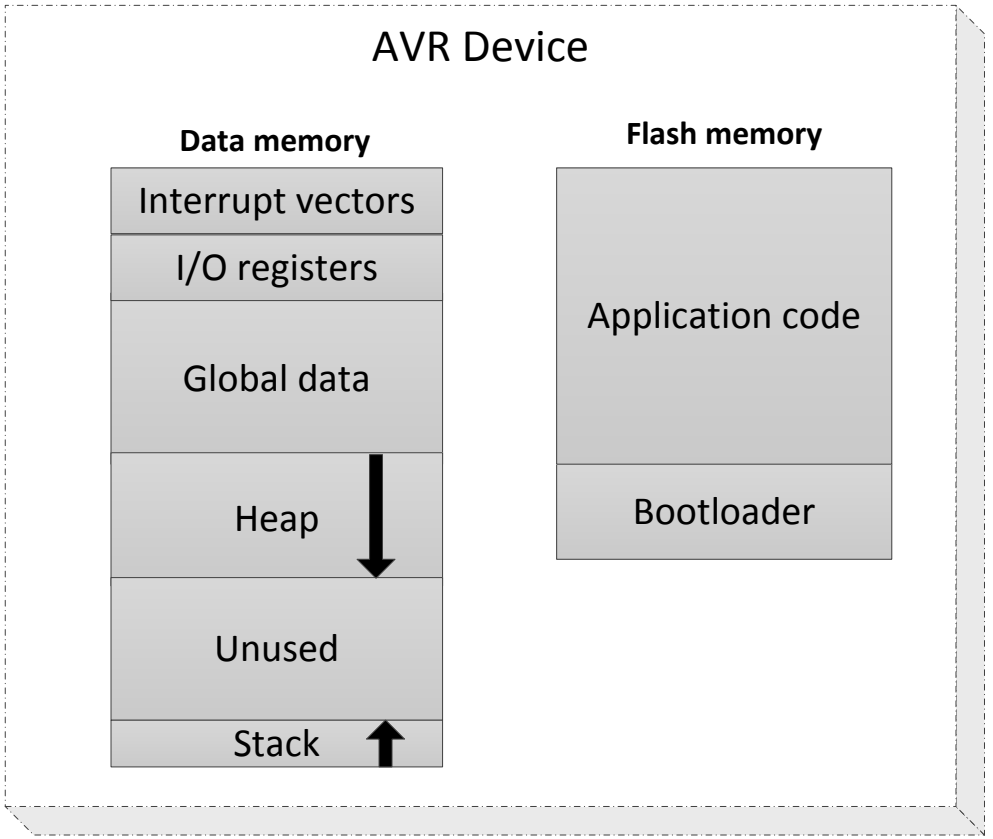
AVR is an architecture used by a widely variety of devices used in the IoT, but its security has not attracted sufficient attention

Outline

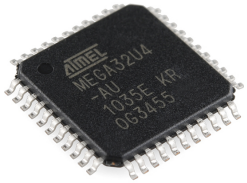
Atmel AVR: Harvard-based architecture



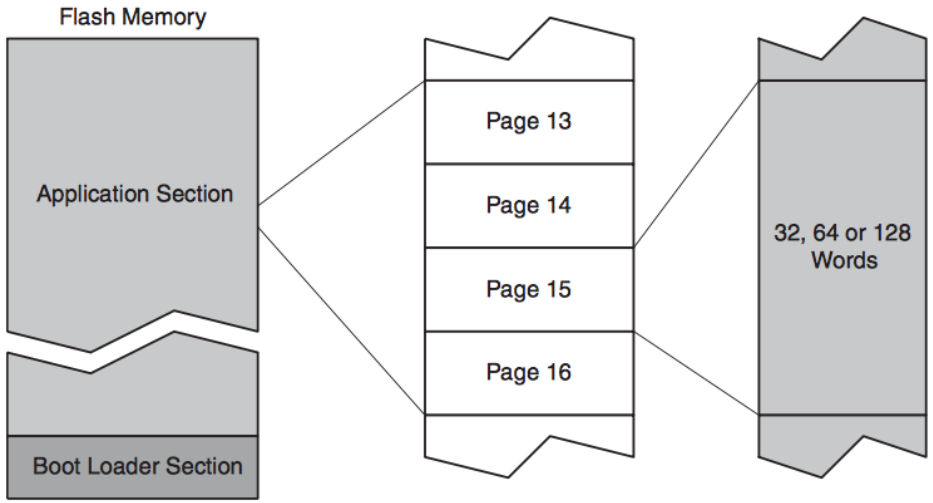
- Code and data memories are physically separated
  - Flash memory: executable, but R/W only from from bootloader
  - SRAM memory: R/W, and not executable



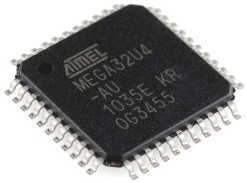
Atmel AVR: Harvard-based architecture



- Code and data memories are physically separated
  - Flash memory: executable, but R/W only from from bootloader
  - SRAM memory: R/W, and not executable
- Flash is organized in pages

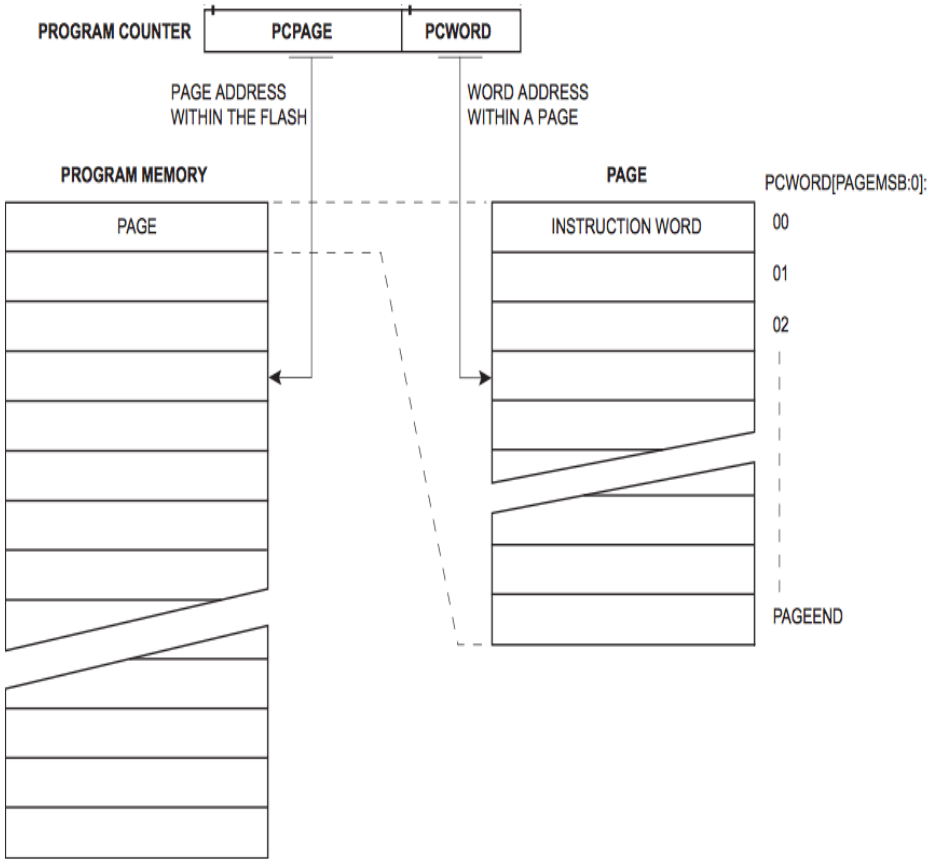


Atmel AVR: Harvard-based architecture



- Code and data memories are physically separated
  - Flash memory: executable, but R/W only from from bootloader
  - SRAM memory: R/W, and not executable
- Flash is organized in pages
  - PC encodes the page number and the offset within a page

$address = PCPage * PSize + PCWord$

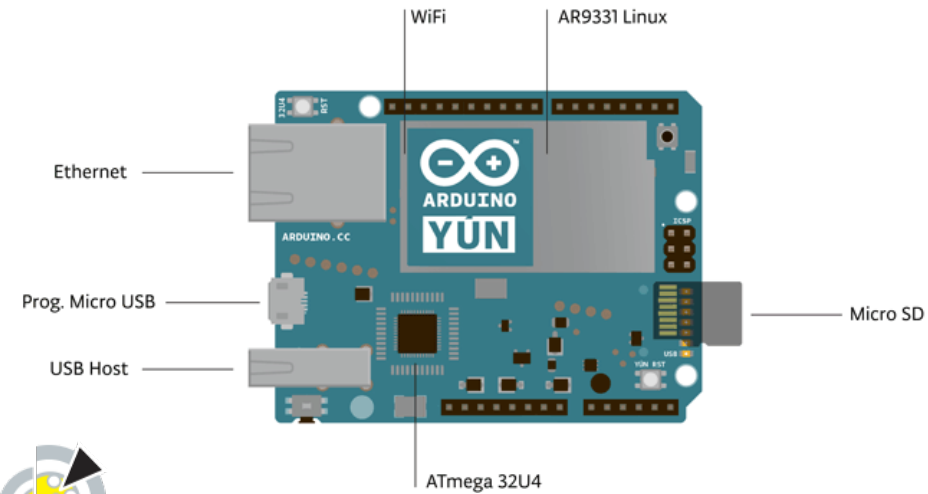


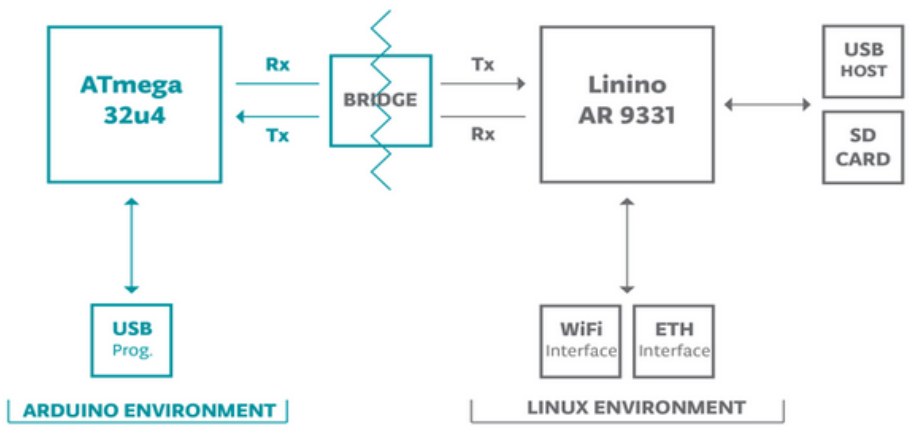
In this work we provide a POC exploit targeting a device named Arduino Yun

Arduino: "open-source electronics platform. Easy-to-use hw and sw"
Arduino Yun: "design connected devices and IoT projects"

Source: Arduino official site

The Yun contains two chips connected through a internal serial port dubbed Bridge





Outline

The proposed exploitation abuses a stack overflow to perform a code reuse attack

Main goal: execute commands in the Openwrt-Yun

Bridge Library → Process → `void runShellCommand (String *cmd);`

```
e26:       81 e5           ldi     r24, 0x51       ; 81
e28:       93 e0           ldi     r25, 0x03       ; 3
e2a:       0e 94 d8 16     call    0x2db0  ; 0x2db0 <Process::runShellCommand(String const&)>
```

[In AVR Arguments are passed through registers (e.g. r24 and r25)]

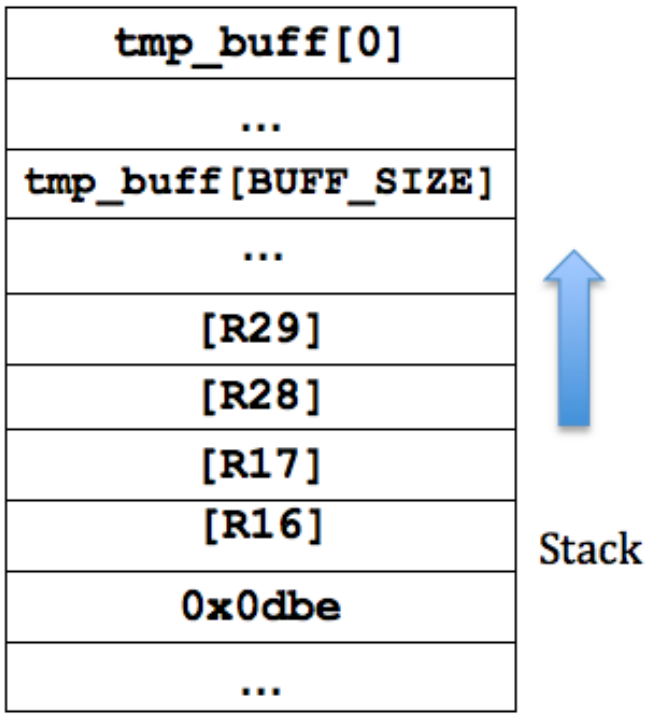
Steps:

1. Hijack the control flow (e.g. stack overflow)
2. ROP to inject the data and prepare the arguments
3. *ret2lib* to force the execution of `runShellCommand`

When a function is called, the return address is stored in the stack

```
dba:        0e 94 ed 05     call    0xbda   ; 0xbda <_Z15vulnerable_funcv>
dbe:        ec 01           movw    r28, r24
```

```
00000bda <_Z15vulnerable_funcv>:
     bda:       0f 93           push    r16
     bdc:       1f 93           push    r17
     bde:       cf 93           push    r28
     be0:       df 93           push    r29
    uint8_t tmp_buff [BUFF_SIZE];
     c4a:       df 91           pop     r29
     c4c:       cf 91           pop     r28
     c4e:       1f 91           pop     r17
     c50:       0f 91           pop     r16
     c52:       08 95           ret
```



A stack overflow vulnerability allows an adversary to hijack the control flow

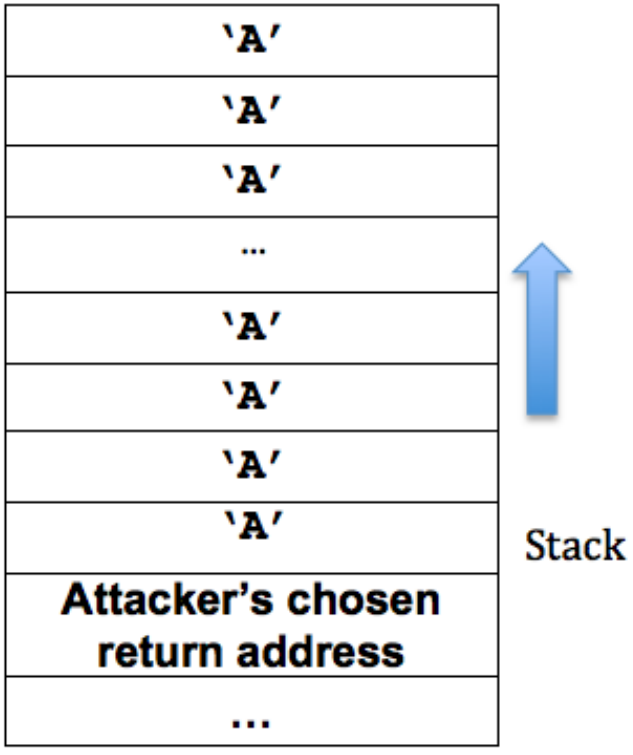
```
dba:        0e 94 ed 05     call    0xbda   ; 0xbda <_Z15vulnerable_funcv>
dbe:        ec 01           movw    r28, r24
```

```
00000bda <_Z15vulnerable_funcv>:
     bda:       0f 93           push    r16
     bdc:       1f 93           push    r17
     bde:       cf 93           push    r28
     be0:       df 93           push    r29
```

[Stack overflow vulnerability]

```
while (BTSerial.available() > 0) {
  char c = BTSerial.read();  //get
  tmp_buff[i] = c;
  i++;
```

```
     c4a:       df 91           pop     r29
     c4c:       cf 91           pop     r28
     c4e:       1f 91           pop     r17
     c50:       0f 91           pop     r16
     c52:       08 95           ret
```



ROP is based on chaining different pieces of code called gadgets to perform the desired operation

VulnerableFunction:

```
00000bda <_Z15vulnerable_funcv>:
     bda:       0f 93           push    r16
     bdc:       1f 93           push    r17
     bde:       cf 93           push    r28
     be0:       df 93           push    r29
```

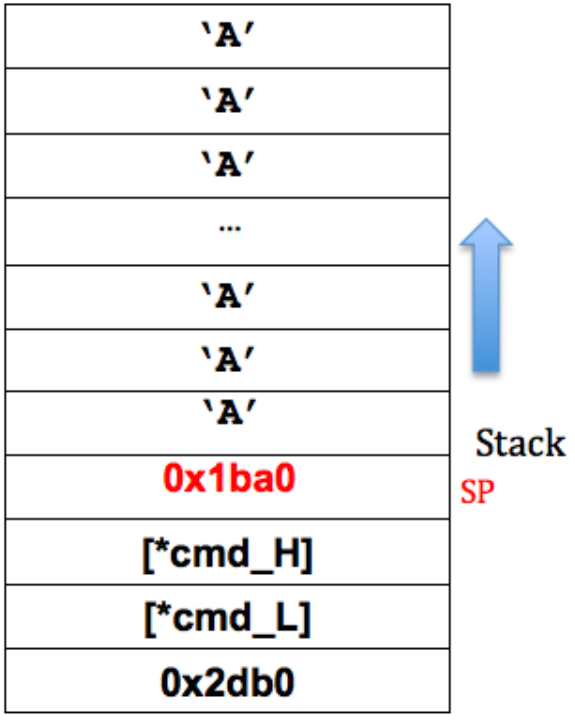
```
     c4a:       df 91           pop     r29
     c4c:       cf 91           pop     r28
     c4e:       1f 91           pop     r17
     c50:       0f 91           pop     r16
PC → c52:       08 95           ret
```

Gadget 1 (LoadArguments):

```
1ba0:       9f 91           pop     r25
1ba2:       8f 91           pop     r24
1bc0:       18 95           reti
```

Gadget 2 (runShellCommand)

```
00002db0 <Process::runShellCommand(String const&)>:
   2db0:       cf 93           push    r28
   2db2:       df 93           push    r29
```



ROP is based on chaining different pieces of code called gadgets to perform the desired operation

VulnerableFunction:

```
00000bda <_Z15vulnerable_funcv>:
     bda:       0f 93           push    r16
     bdc:       1f 93           push    r17
     bde:       cf 93           push    r28
     be0:       df 93           push    r29
```

```
     c4a:       df 91           pop     r29
     c4c:       cf 91           pop     r28
     c4e:       1f 91           pop     r17
     c50:       0f 91           pop     r16
     c52:       08 95           ret
```

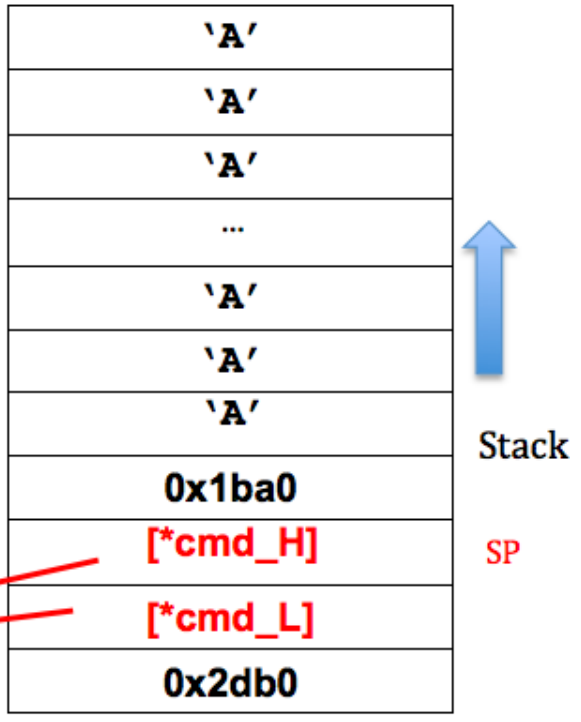
Gadget 1 (LoadArguments):

PC →

```
1ba0:       9f 91           pop     r25
1ba2:       8f 91           pop     r24
1bc0:       18 95           reti
```

Gadget 2 (runShellCommand)

```
00002db0 <Process::runShellCommand(String const&)>:
   2db0:       cf 93           push    r28
   2db2:       df 93           push    r29
```



ROP is based on chaining different pieces of code called gadgets to perform the desired operation

VulnerableFunction:

```
00000bda <_Z15vulnerable_funcv>:
     bda:       0f 93           push    r16
     bdc:       1f 93           push    r17
     bde:       cf 93           push    r28
     be0:       df 93           push    r29
```

```
     c4a:       df 91           pop     r29
     c4c:       cf 91           pop     r28
     c4e:       1f 91           pop     r17
     c50:       0f 91           pop     r16
     c52:       08 95           ret
```

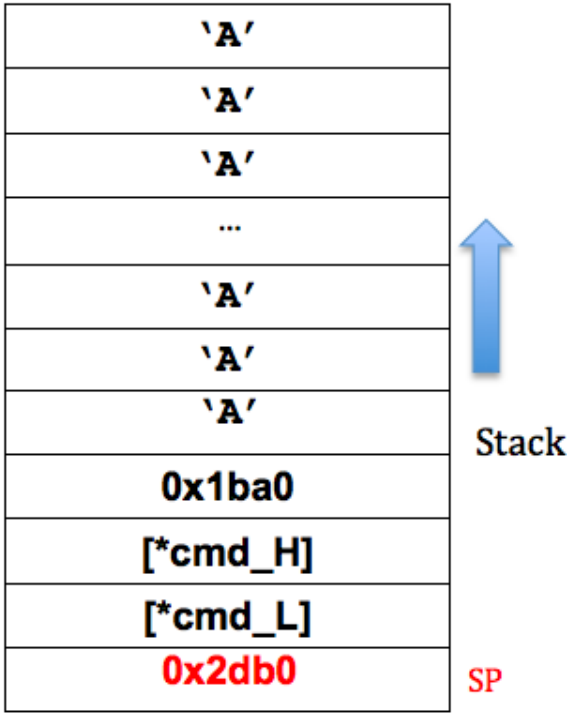
Gadget 1 (LoadArguments):

```
1ba0:       9f 91           pop     r25
1ba2:       8f 91           pop     r24
1bc0:       18 95           reti
```

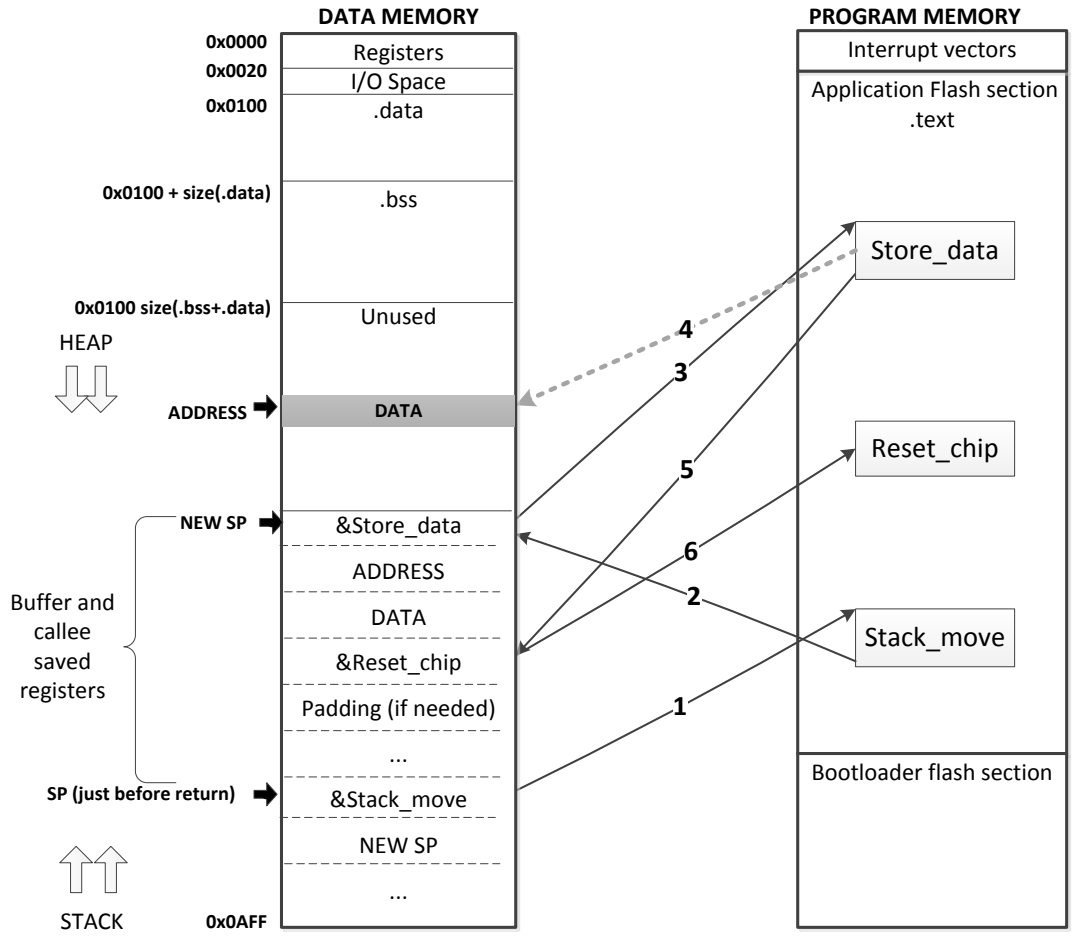
PC →

Gadget 2 (runShellCommand)

```
00002db0 <Process::runShellCommand(String const&)>:
   2db0:       cf 93           push    r28
   2db2:       df 93           push    r29
```



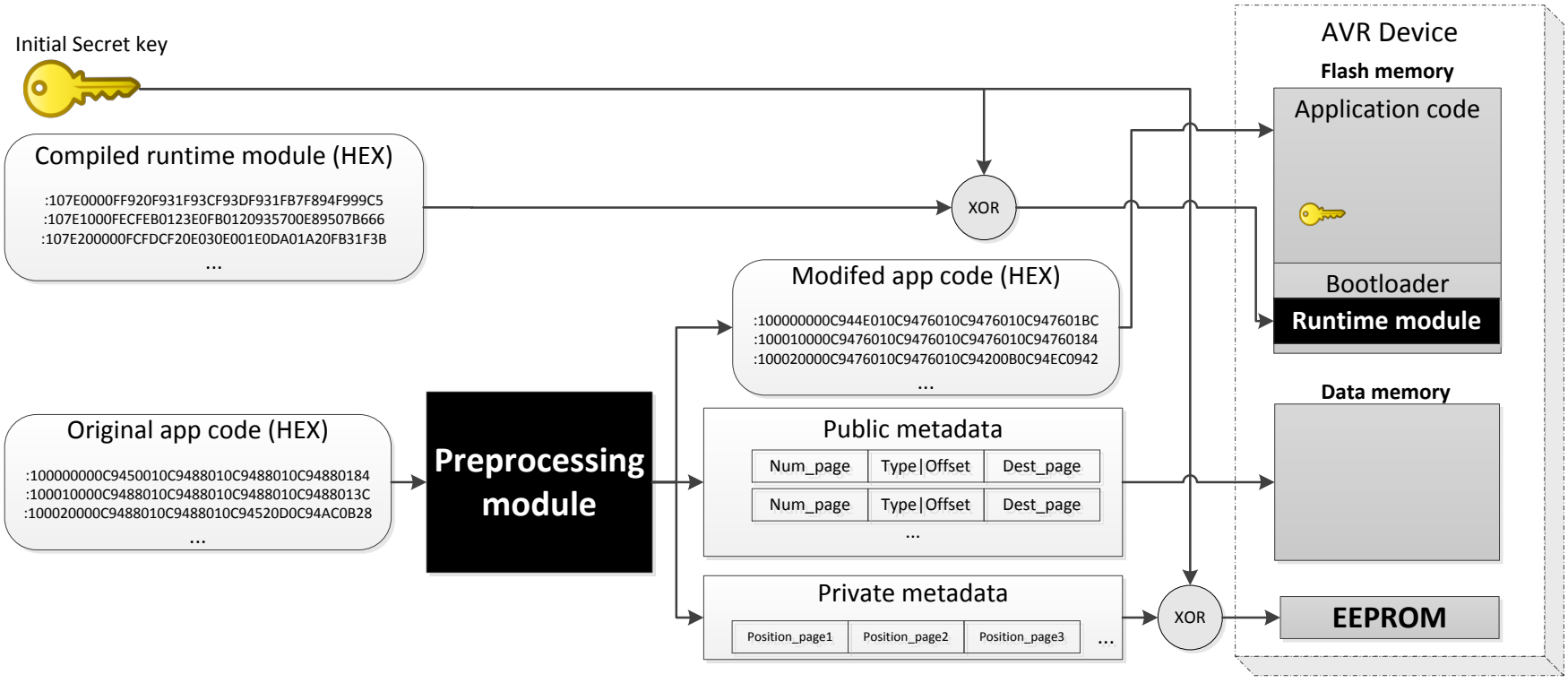
Prior to calling the function, it is needed to inject the command (data in unused SRAM)



Adapted from: [Francillon & Castellucia, 2008] and [Habibi et al., 2015]

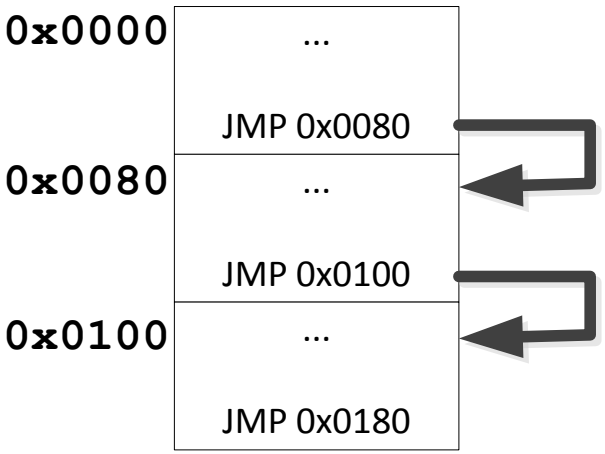
Outline

AVRAND first preprocesses the binary being flashed, and then applies randomization at runtime



The preprocessing module converts all relative pointers to absolute and link pages

- Convert all relative to absolute pointers
  - e.g. RCALL $\rightarrow$ CALL and RJMP $\rightarrow$ JMP
- Page linking through direct jumps



The preprocessing module outputs information required for the runtime randomization

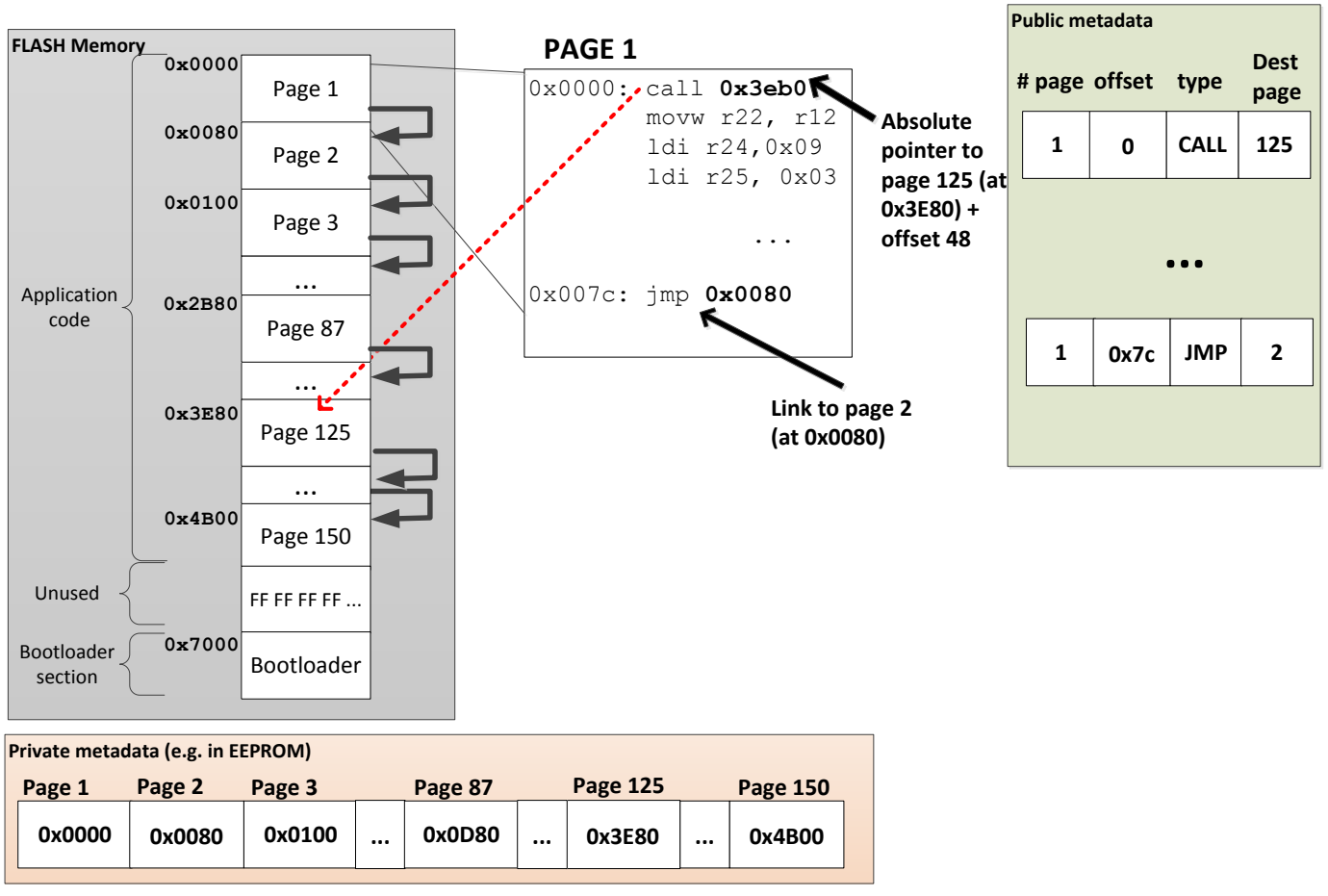
- **Public metadata:** where are the pointers?
  - Required to recalculate pointers after randomization

Src. page	Offset	Type	Dest. page

  - PC addresses encode page address (dynamic) and offset (static)
- **Private metadata:** where are the pages?
  - Required to know the page addresses within the flash
  - List of addresses indexed by the page number

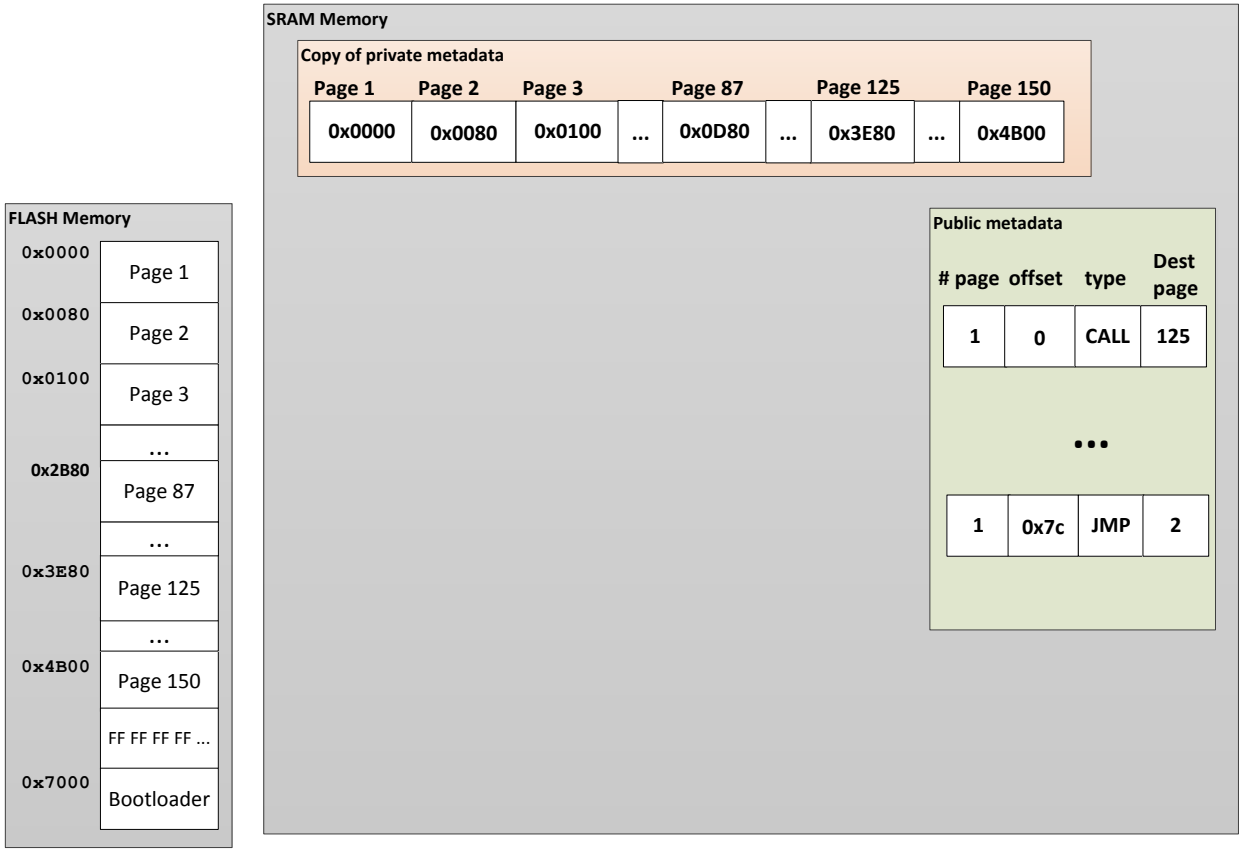
AVRAND explained

Initial layout



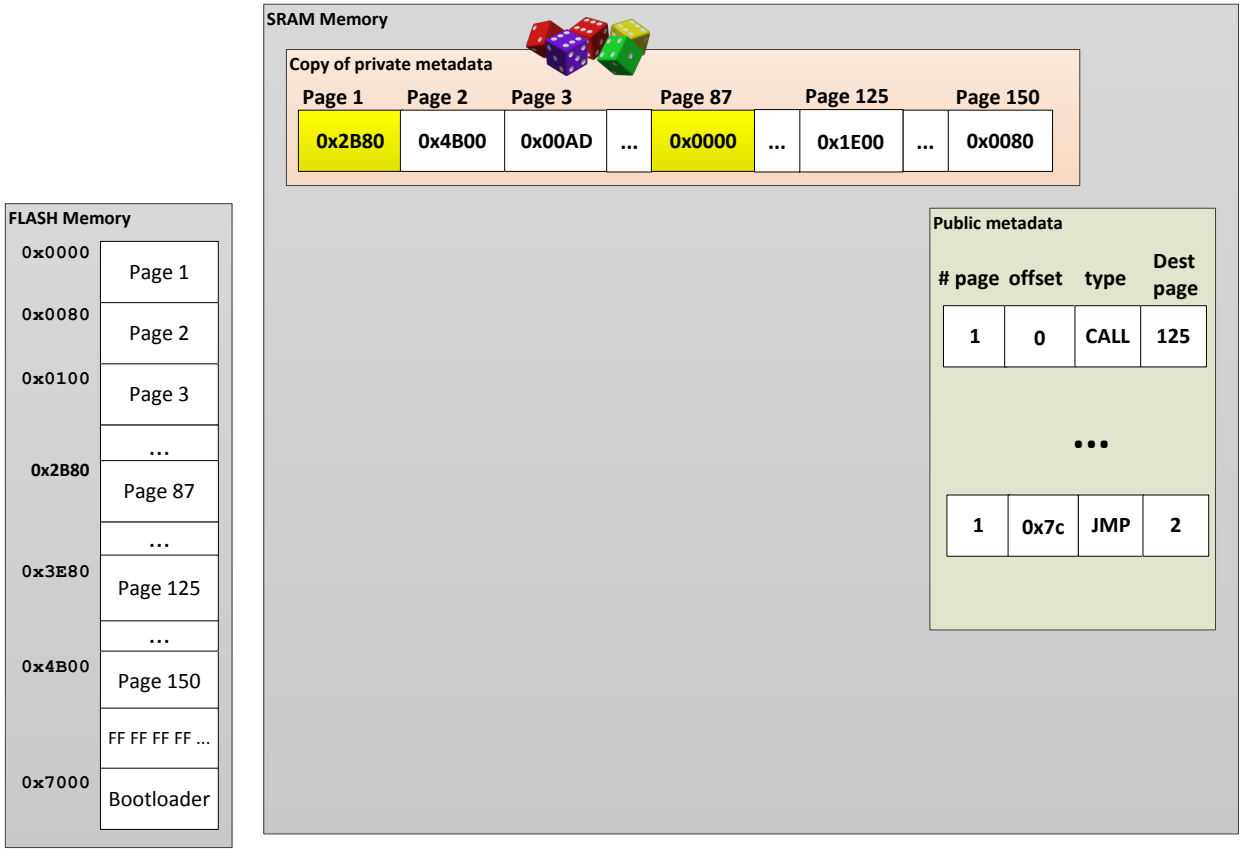
AVRAND explained

1. Copy the private metadata to SRAM



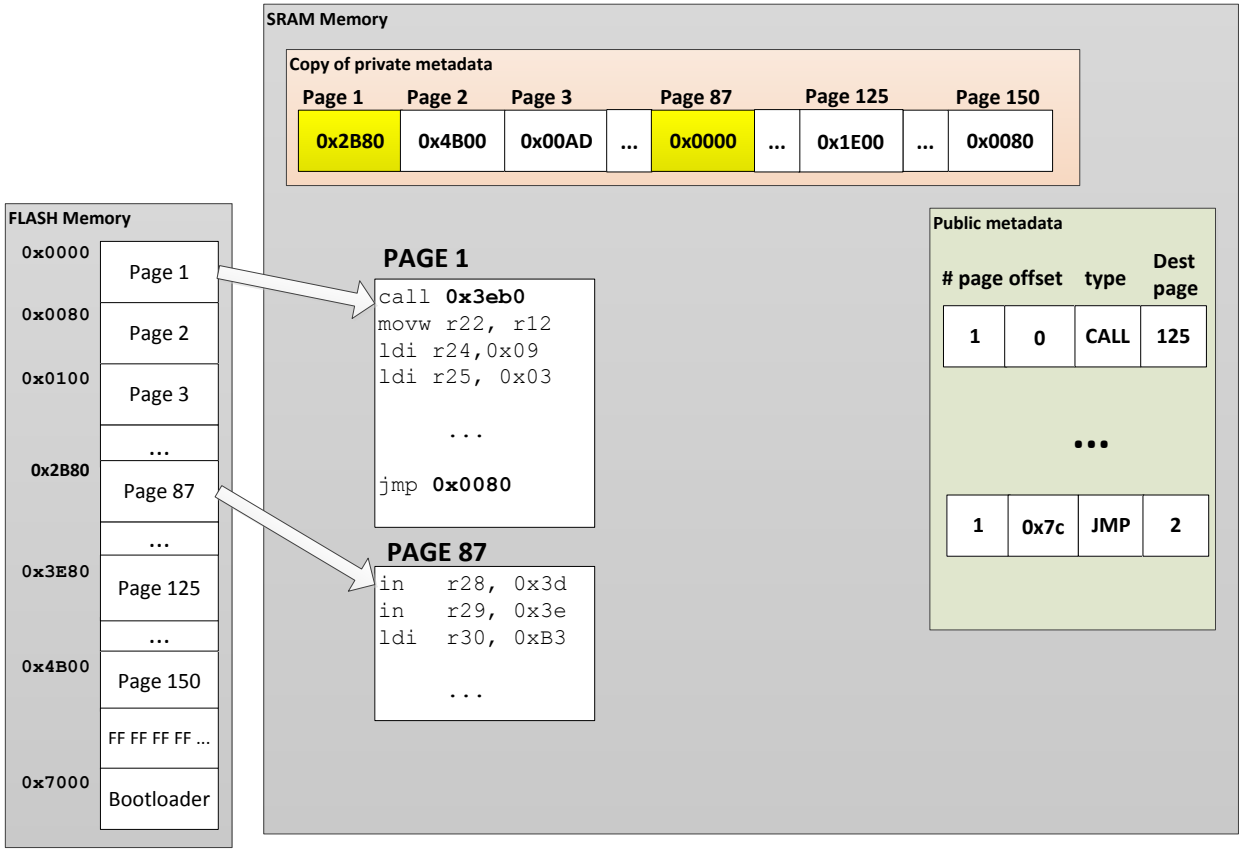
AVRAND explained

2. Modify the copy by swapping pages randomly



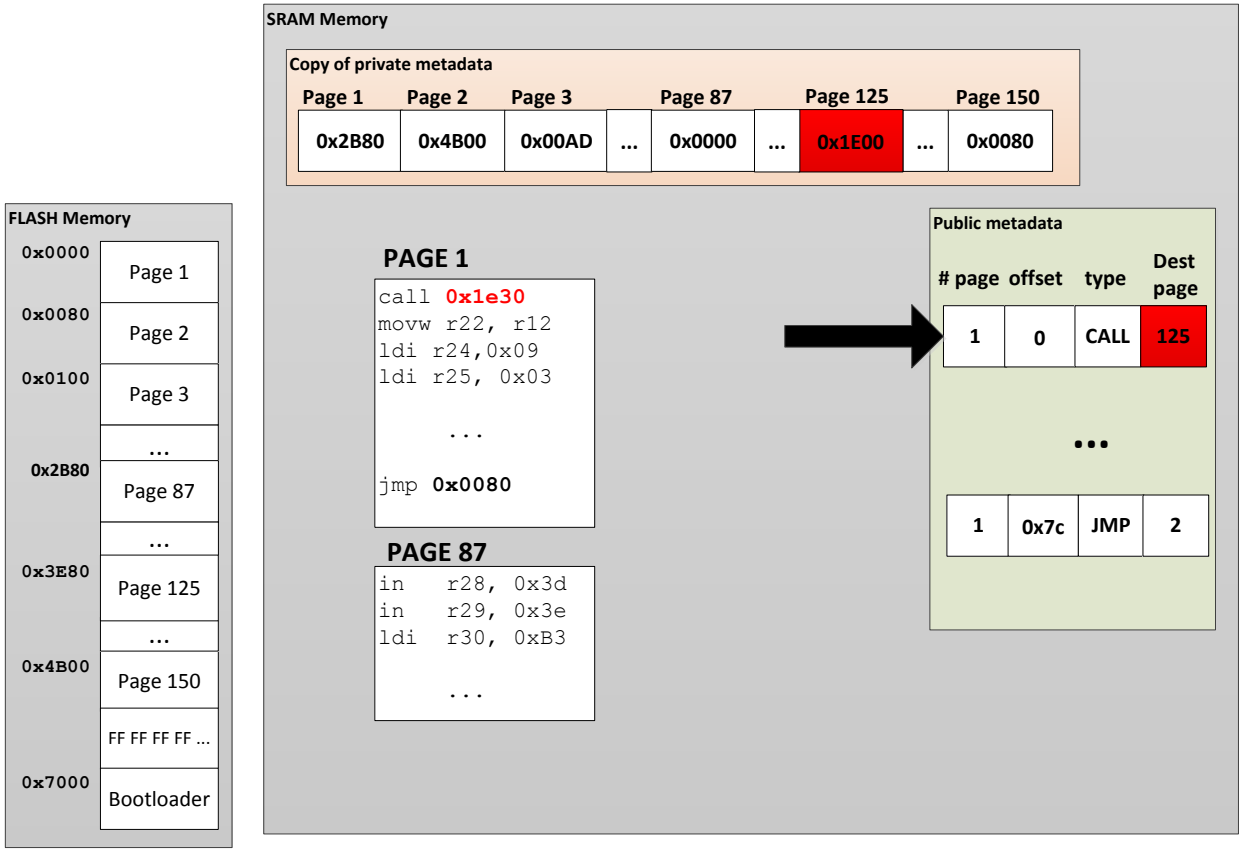
AVRAND explained

3. Copy each pair of swapped pages to SRAM



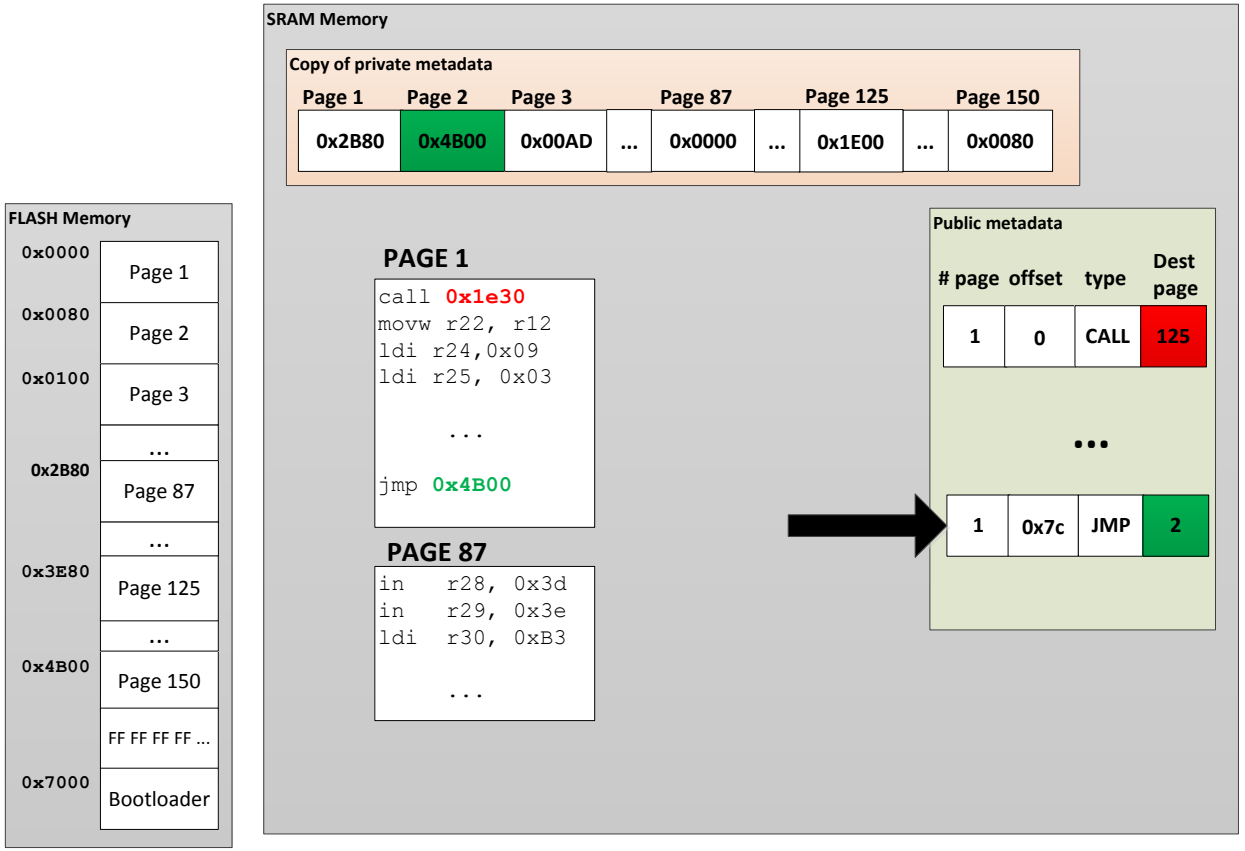
AVRAND explained

4. Update pointers on each page (using the metadata)



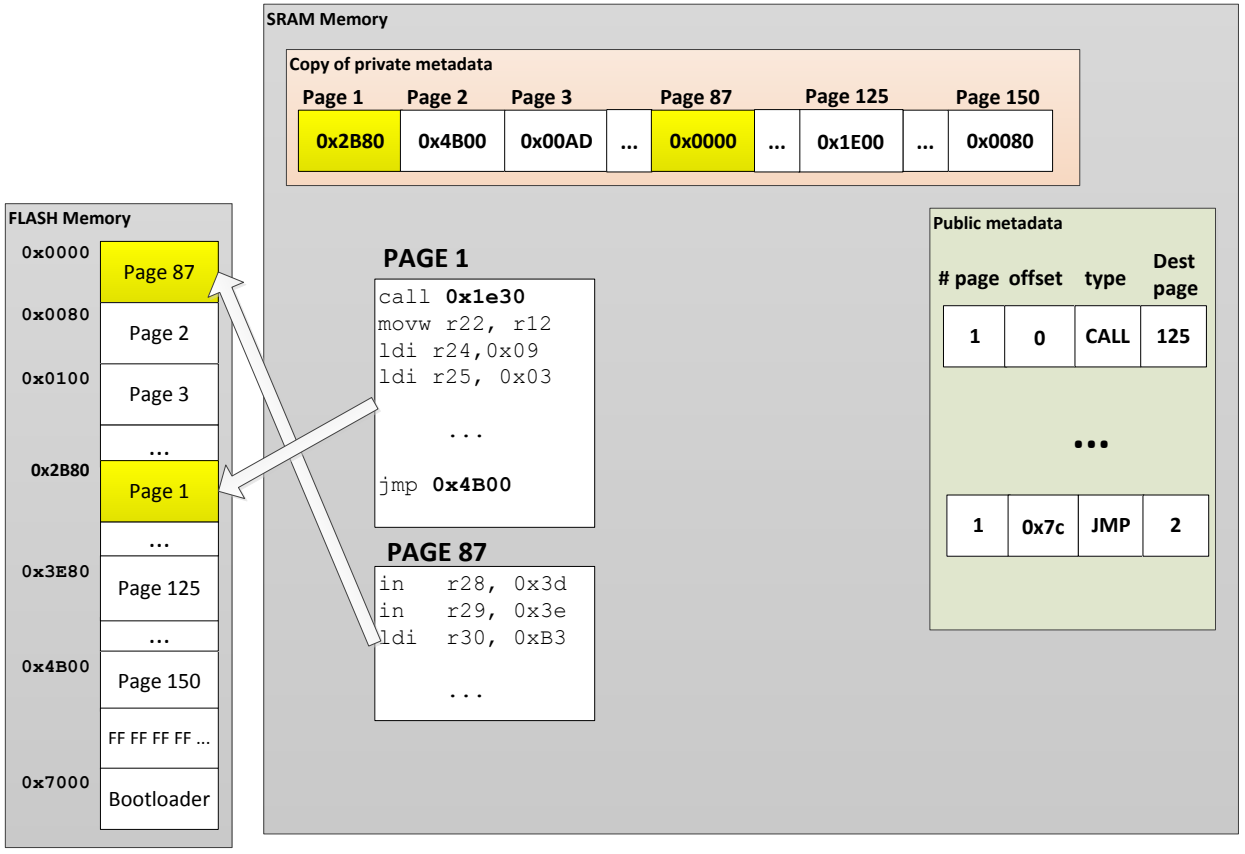
AVRAND explained

4. Update pointers on each page (using the metadata)



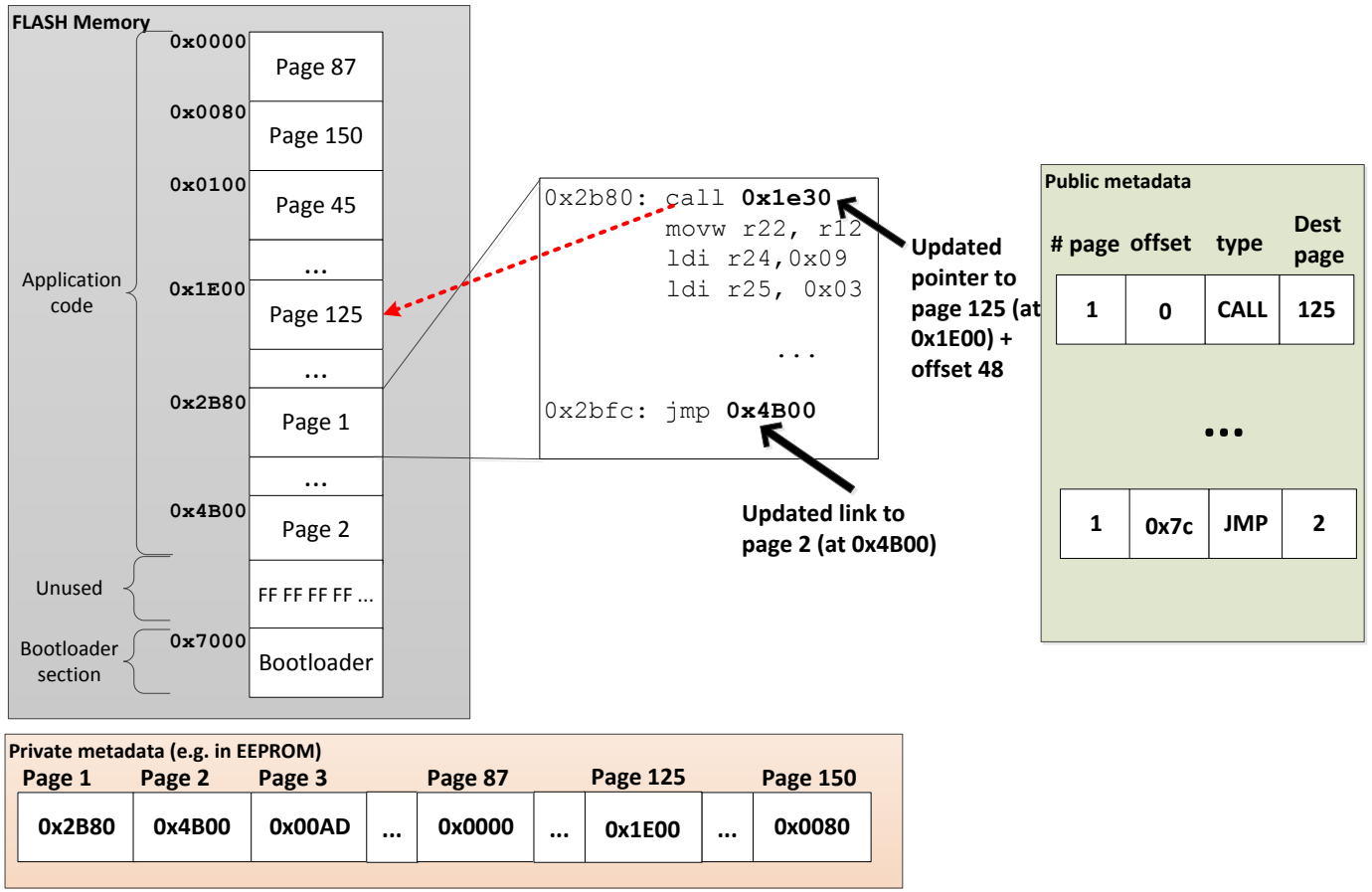
AVRAND explained

5. Copy back to flash into each other's previous position



AVRAND explained

Modified layout



The bootloader itself must be protected from code reuse attacks

Goal: Obfuscating the bootloader

- Approach: the bootloader is stored encrypted, and is decrypted at runtime.
- Due to resource limitations, we use XOR-based encryption
  - Brute force prevention: Key is renewed during each re-randomization
- Steps
  1. Decrypt bootloader
  2. Jump to randomization engine
  3. Renew key
  4. Encrypt bootloader
  5. Jump to the beginning of the program (Entry Point)

Outline

AVRAND goes one step further regarding security of AVR-based embedded devices

- AVR is an architecture used in many devices, but its security has not been considered
- A POC exploit shows how an Arduino chip can be compromised using code reuse attacks
- AVRAND hinders these attacks by means of memory layout randomization
- Strengths:
  - Software-based defense (independent of manufacturers, reduce costs)
  - Insignificant processing overhead ($<$1s)
  - High entropy (though depends on the number of pages)
- Limitations:
  - Extra memory overhead ($\sim$ 20%)
  - Reduction of device lifetime (limited flash cycles)

Eskerrik asko!

spastran@inf.uc3m.es

Prototypes available at:

`http://www.seg.inf.uc3m.es/~spastran/avrand`

AVRAND: A Software-Based Defense Against Code Reuse Attacks for AVR Embedded Devices

Sergio Pastrana, Juan Tapiador,
Guillermo Suarez-Tangil, Pedro Peris-Lopez

DIMVA 2016
San Sebastian. 7,8 July 2016

Backup slide: AVRAND limitations

Randomization can occur in each device reset or periodically:

- Frequency of randomizations: depends on the scenario
  - On each device reset (current approach)
    - ✓ Prevent code reuse attacks that crash the device
    - ✗ May be vulnerable to brute-force that clean the stack
  - Periodically, using timeouts
    - ✓ Brute force attacks are restricted to a limited period of time
    - ✗ Still, there is a vulnerable window
    - ✗ Flash memory has limited re-flashing cycles (e.g. 4.000 in Yun)
- Limitation: Code size overhead
  - The preprocessing module increases code size by an avg. of 20%
  - Binaries are compiled with full optimization enabled (*-O3*)
  - All sample sketches from Arduino official site fit well in the Yun device, though