Graduate School of Information Sciences, TOHOKU University

17 September 2015

Buffer Overflow Attack with Multiple Fault Injection and a Proven Countermeasure

PROOFS2015, Saint Malo, France

Shoei Nashimoto, Naofumi Homma,
Yu-ichi Hayashi and Takafumi Aoki

Tohoku University, Japan

Embedded devices become attractive

- Increase attractions to attack embedded systems
  - Many devices connect to networks (Internet-of-Things)
  - Worth paying high cost, e.g., attacks to cryptographic hardware
- Fault injection attacks
  - Inject fault(s) in cryptographic operation, and obtain secret key from faulty output(s)
  - Fault injection into microcontrollers often brings bit inversion or instruction skip [Agoyan 2010], [Endo 2014]

It is possible to apply fault injection techniques to general-purpose software

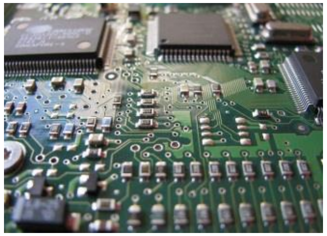
Fault injection attacks to general-purpose software

- Previous works
  - Execute arbitrary code in Java Virtual Machine by inverting bits [Govindavajhala 2003], [Bouffard 2011]
  - Cause effect like buffer overflow (BOF) using instruction skip [Fouque 2012]
- **Not only** cryptographic software
  - Fault injection attacks are also threat to general-purpose software

This work

- Propose buffer overflow attack with multiple fault injection
  - Instruction skips are **not** considered in most software
    - Can invalidate countermeasures by secure coding
  - Overcome typical software countermeasure and perform general buffer overflow (BOF) attack
- Propose effective countermeasure and prove its validity

Attacks to hardware



Side-channel attacks,
Fault injection attacks

+

Attacks to software

DoS attacks,
BOF attacks,
Injection attacks

Outline

- Background
- Buffer overflow attacks
- Proposed attack & experiment
- Countermeasure
- Conclusion

What are BOF attacks?

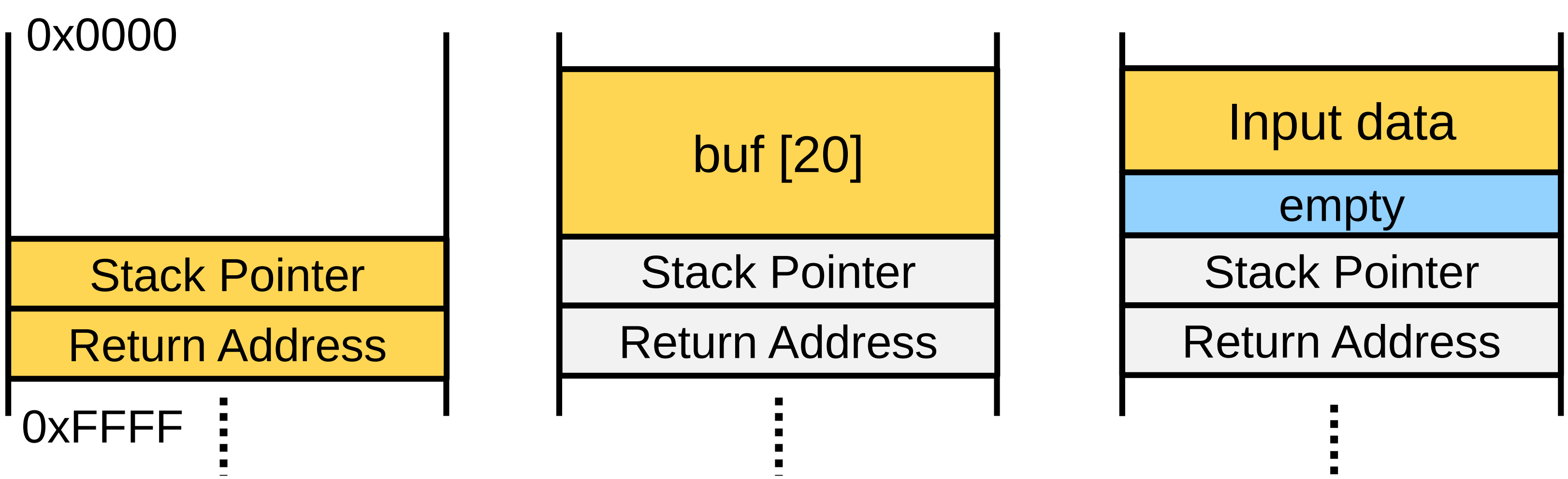
- Buffer overflow (BOF)
  - Invalid memory overwrite caused by input that exceeds assigned memory size
  - Commonly happen when using flexible languages that can finely handle a memory region, such as C/C++
    - *strcpy()*, *scanf()*, *gets()* are dangerous
- BOF attacks
  - Use BOF vulnerability to execute malicious operations
    - Abnormal stop of OS or applications
    - Gaining administrator rights

How to perform BOF attack (e.g., strcpy())

```
int main(int argc, char *argv[]) {
① sub_function(data);

  return 0;
}
```

```
void sub_function(char *data) {
② char buf[20];

③ strcpy(buf, data);

  return;
}
```

① Function call ② Memory allocation ③ String-copy operation

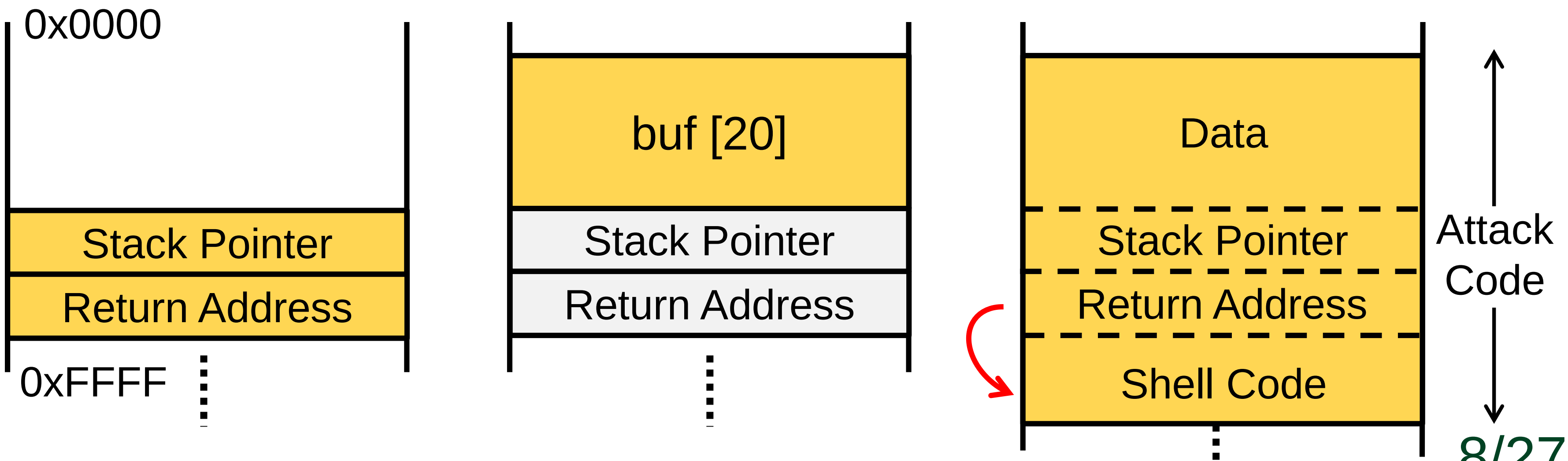


How to perform BOF attack (e.g., strcpy())

```
int main(int argc, char *argv[]) {
①  sub_function(data);

    return 0;
}
```

```
void sub_function(char *data) {
②  char buf[20];

③  strcpy(buf, data);

    return;
}
```

① Function call ② Memory allocation ③ String-copy operation



Countermeasure against BOF attacks

Name	Method	Layer
ASLR (Address Space Layout Randomization)	Change stack address for every execution.	OS (Operating System)
SG (Stack Guard)	Add random numbers and check them at the end of function.	Compiler
DEP (Data Execution Prevention), ES (Exec Shield)	Prohibit execution of all code in the stack.	OS, CPU, Compiler
ISL (Input Size Limitation)	Use function that can limit input size.	Program

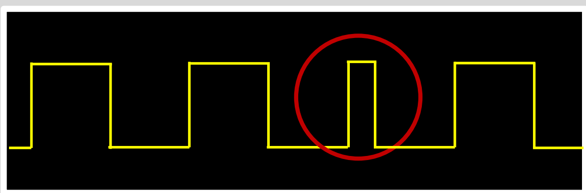
- Input Size Limitation (ISL)
  - Only need standard C library
  - Simply change function to use
    - *strcpy(dest, src) → strncpy(dest, src, size)*

Outline

- Background
- Buffer Overflow Attacks
- Proposed attack & Experiment
- Countermeasure
- Conclusion

Concept of the proposed attack

- Assumption
  - Feeding glitchy clock signal into CPU enables instruction skip(s).



Glitchy clock signal

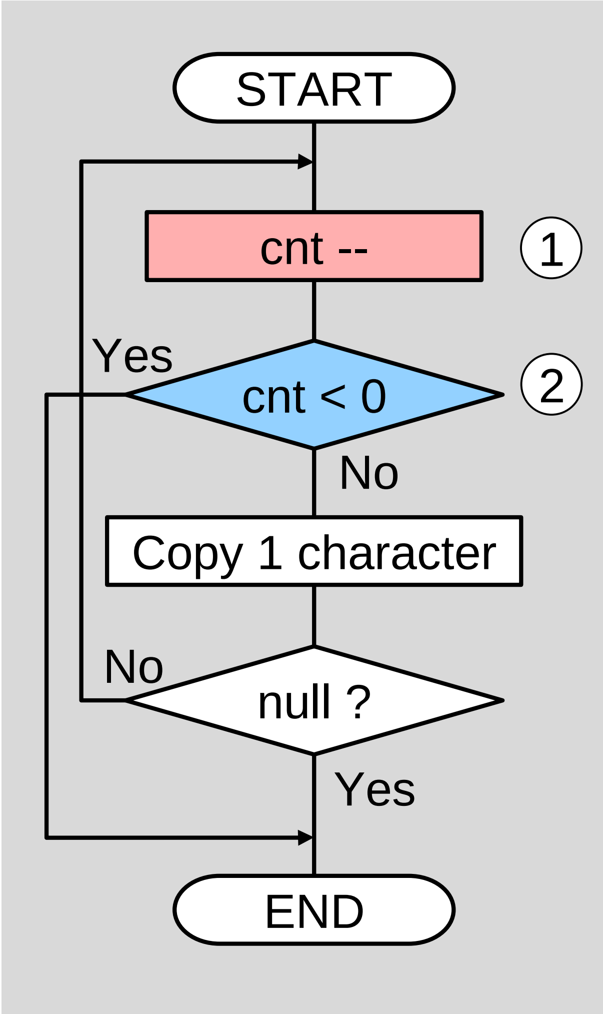
PUSH	R29	Push register on stack
PUSH	R28	Push register on stack
~~RCALL~~	~~PC+0x0001~~	~~Relative call subroutine~~
~~RCALL~~	~~PC+0x0001~~	~~Relative call subroutine~~
PUSH	R0	Push register on stack
~~IN~~	~~R28,0x3D~~	~~In from I/O location~~
IN	R29,0x3E	In from I/O location

Skip multiple and arbitrary instructions

- Attack method
  - Skip a few instructions while input attack code, and invalidate boundary check to make buffers overflowed
  - Take control of CPU like common attacks

BOF using instruction skips

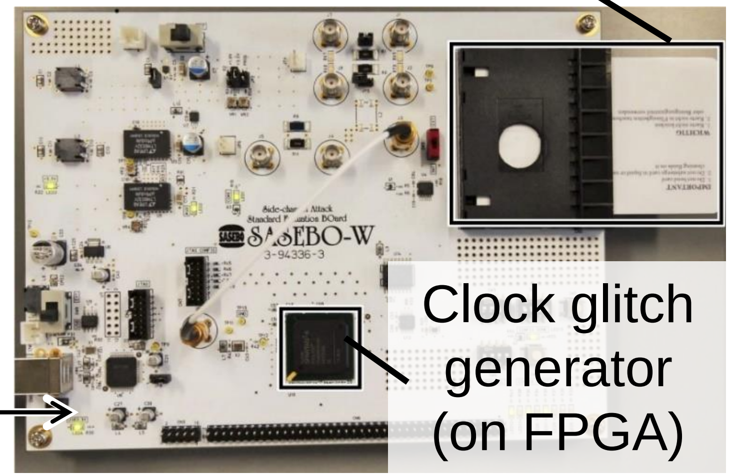
- Target function
  - *strncpy(dst, src, size)*
  - Limit input size up to *size*
- Target instructions
  - ① Subtract (update) instruction
    Continue the loop without update
    → Increase input size by 1 character
  - ② Branch instruction
    Continue the loop unconditionally
    → Over the assigned size



Simplified flow of *strncpy()*

Experimental setup

- Equipments
  - SASEBO-W (Side-channel Attack Standard Evaluation BOard)
  - Smart card (AVR ATmega163)
  - PC

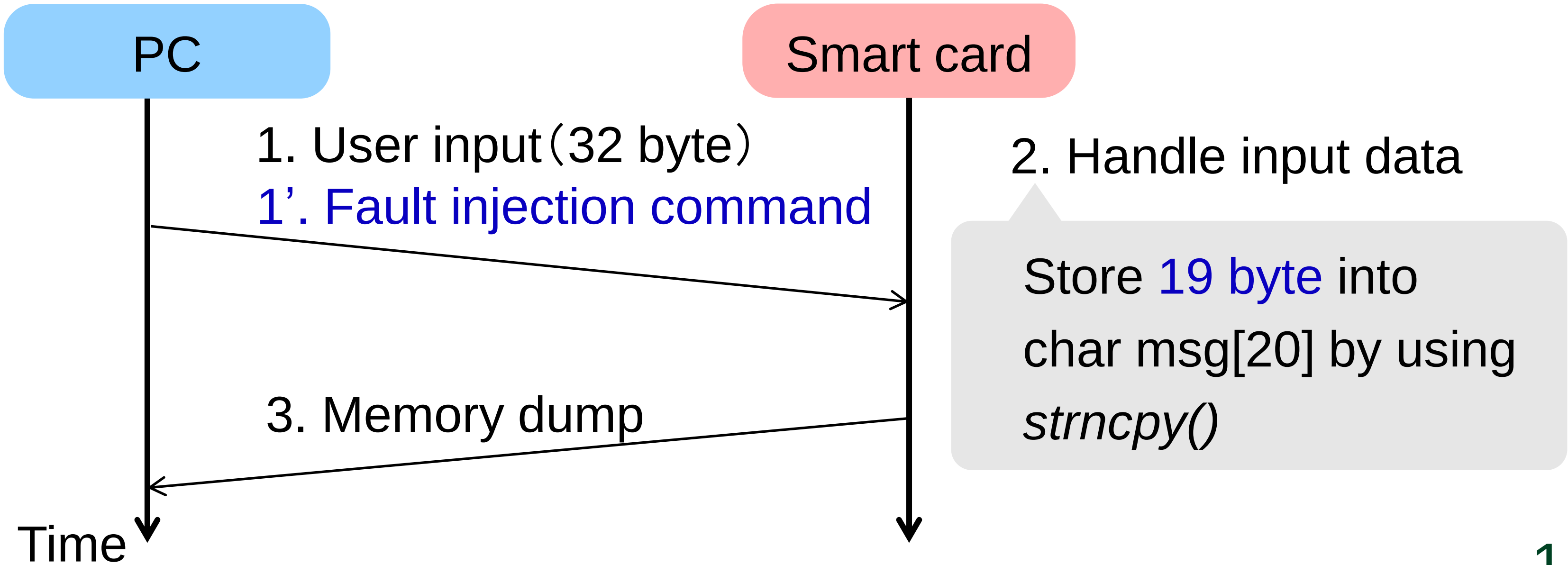


- Conditions

Microcontroller	AVR ATmega163 (8 bit)
Clock frequency of microcontroller	Up to 8 MHz
Compiler	avr gcc (4.3.3) (Not optimized by -o0)
FPGA	Xilinx XC6SLX150-FGG484
Attack condition	Program is known

Experimental outline

- Procedure
  - Invalidate the countermeasure of *strncpy()* and perform BOF attack
  - Overwrite return address to call function in the program
- Control flow



Result

- No fault, “A...A” (20 byte)

char msg[20]

```
0100 | 41 41 41 41 41 41 41 41 | AAAAAAAA
0108 | 41 41 41 41 41 41 41 41 | AAAAAAAA
0110 | 41 41 41 00 38 04 03 86 | AAA.8
0118 | 3c 04 06 0b 00 bf 00 53 |
0120 | 00 00 00 00 00 00 00 00 | .
0128 | 00 00 00 00 00 00 00 00 | .
```

Stack Pointer
+
Return Address

- 5 faults, “A...A (0x38) (0x04) (0x04) (0x08)” (24 byte)

```
void hello_world() {  memcpy((char*)0x120, “hello world!”, 12);  }
```

Function maliciously called by BOF attack

(The address is got by the object dump of the program)

Result

- No fault, “A...A” (20 byte)

char msg[20]

```
0100 | 41 41 41 41 41 41 41 41 | AAAAAAAA
0108 | 41 41 41 41 41 41 41 41 | AAAAAAAA
0110 | 41 41 41 00 38 04 03 86 | AAA.8
0118 | 3c 04 06 0b 00 bf 00 53 |
0120 | 00 00 00 00 00 00 00 00 | .
0128 | 00 00 00 00 00 00 00 00 | .
```

Stack Pointer
+
Return Address

- 5 faults, “A...A (0x38) (0x04) (0x04) (0x08)” (24 byte)

```
0100 | 41 41 41 41 41 41 41 41 | AAAAAAAA
0108 | 41 41 41 41 41 41 41 41 | AAAAAAAA
0110 | 41 41 41 41 38 04 04 08 | AAAA8...
0118 | 3c 04 06 0b 00 bf 00 53 | <......S
0120 | 48 65 6c 6c 6f 20 77 6f | Hello wo
0128 | 72 6c 64 21 00 00 00 00 | rld!....
```

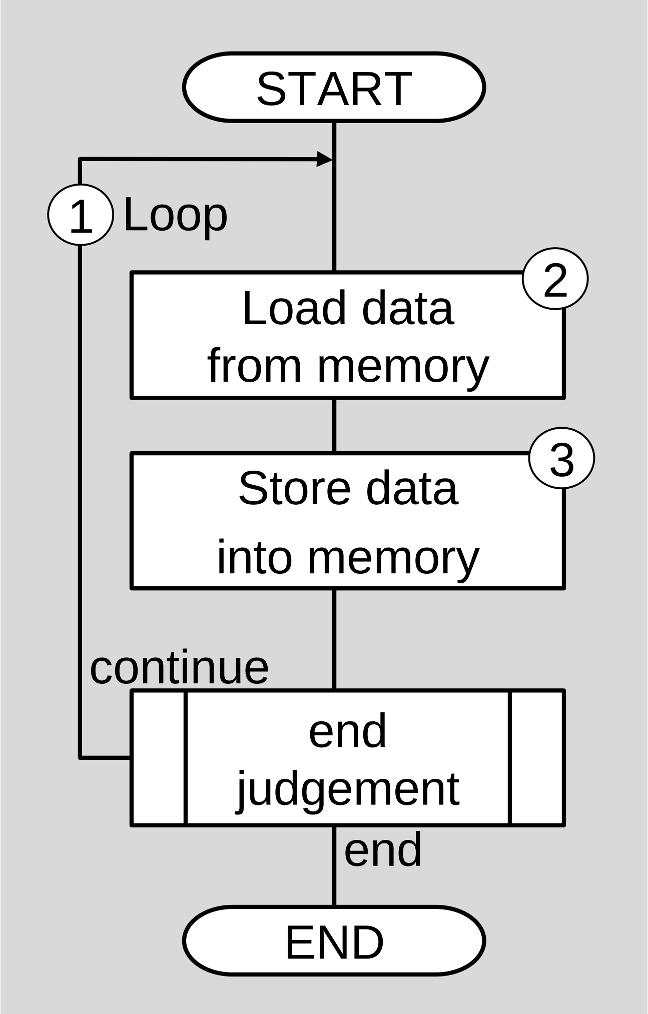
Possibility of proposed attack

- Target functions
  - User functions that have iteration
  - *strncpy(), fgets(), strncmp(), memcpy()*

CERT/CC Top 10 Secure Coding Practices [1]

No1: Validate input.

- Attack conditions
  - Physically accessible (fault injection)
  - Program is known (BOF attack)

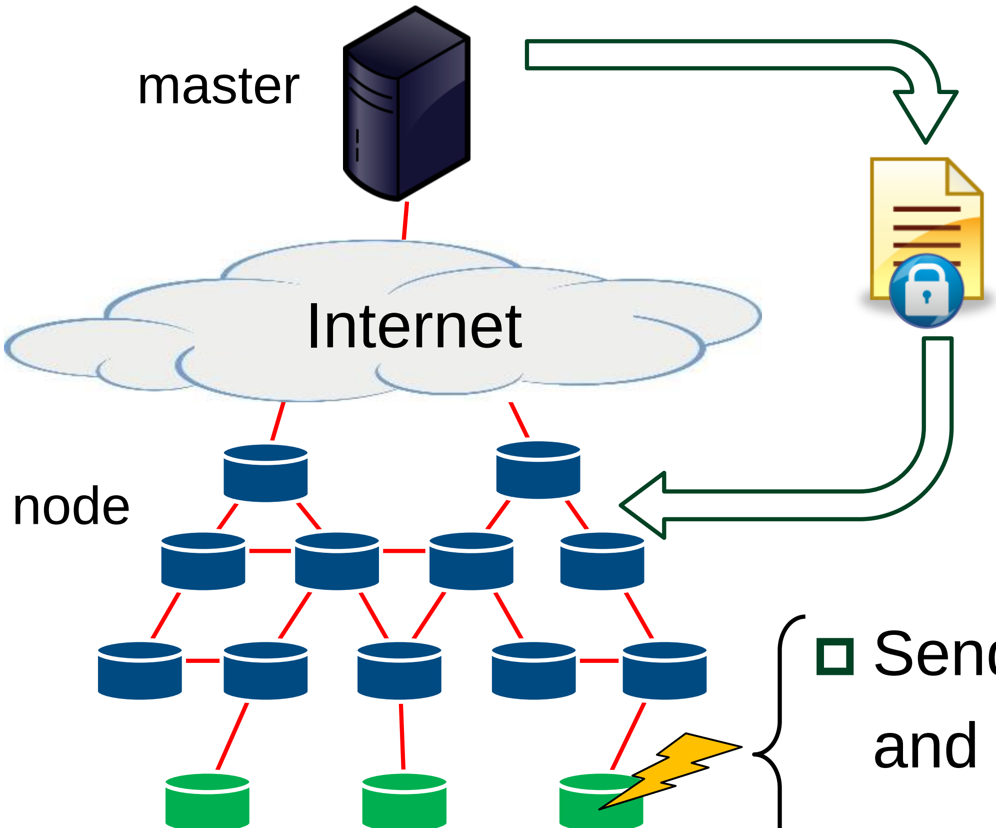


Vulnerable structure

[1] https://www.securecoding.cert.org/confluence/display/seccode/Top+10+Secure+Coding+Practices

Example of attack scenario

- Malicious firmware update in M2M network



Machine To Machine (M2M) sensor network

1. Sign update program by private key
2. Check validity of the program by public key

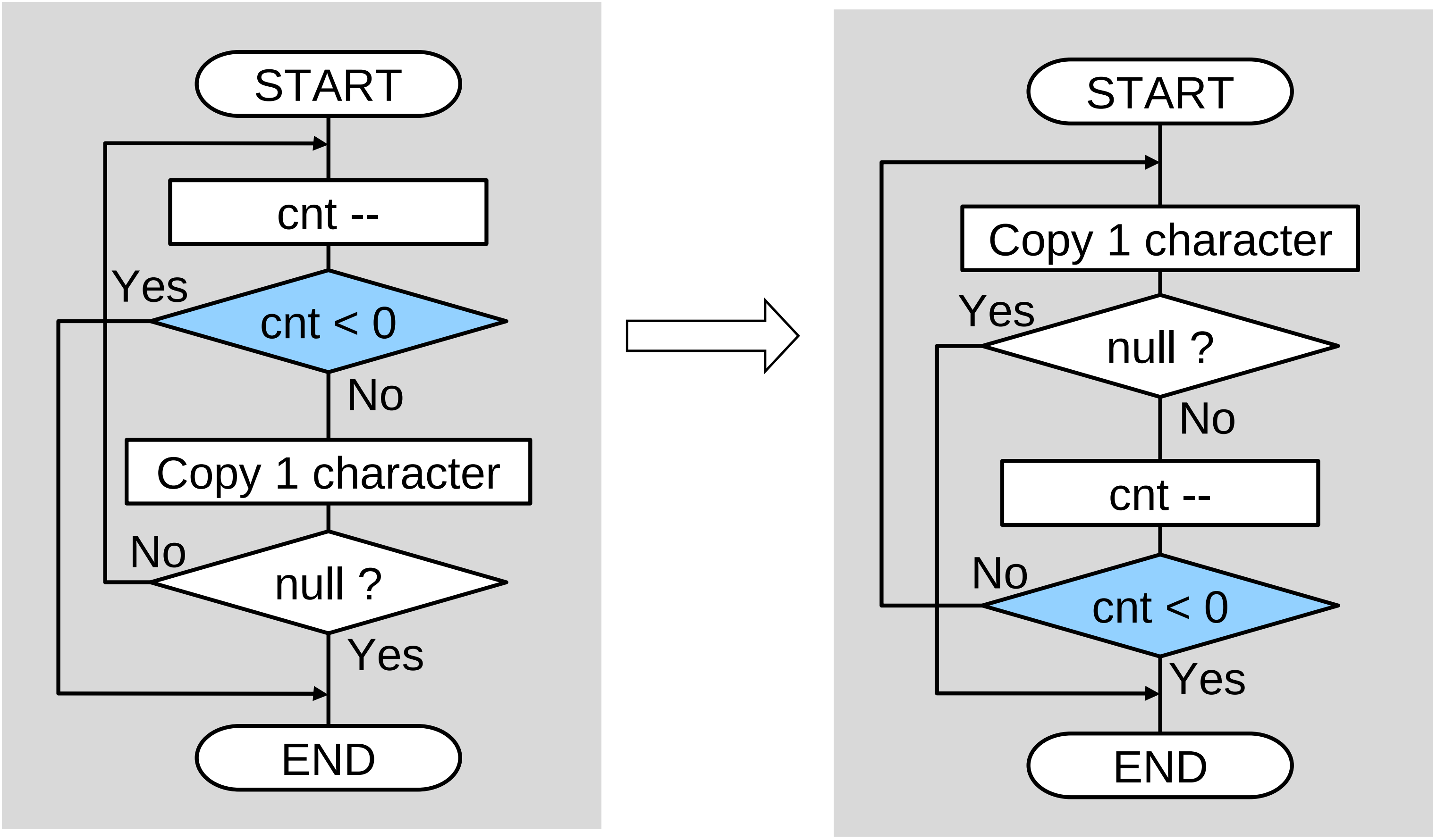
- Send malicious update program, and perform BOF attack
- Directly call update function

Outline

- Background
- Buffer Overflow Attacks
- Proposed attack & Experiment
- Countermeasure
- Conclusion

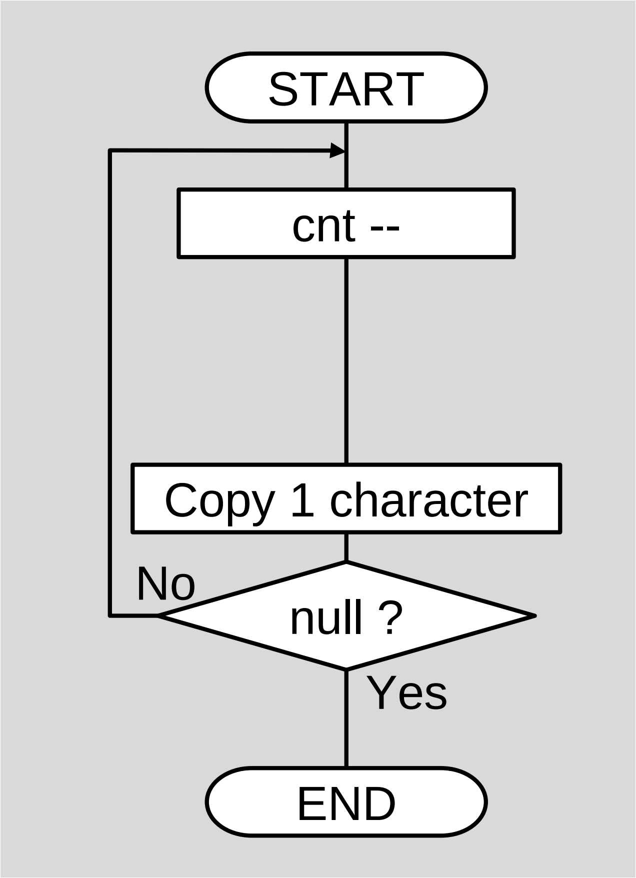
How to protect branch instruction

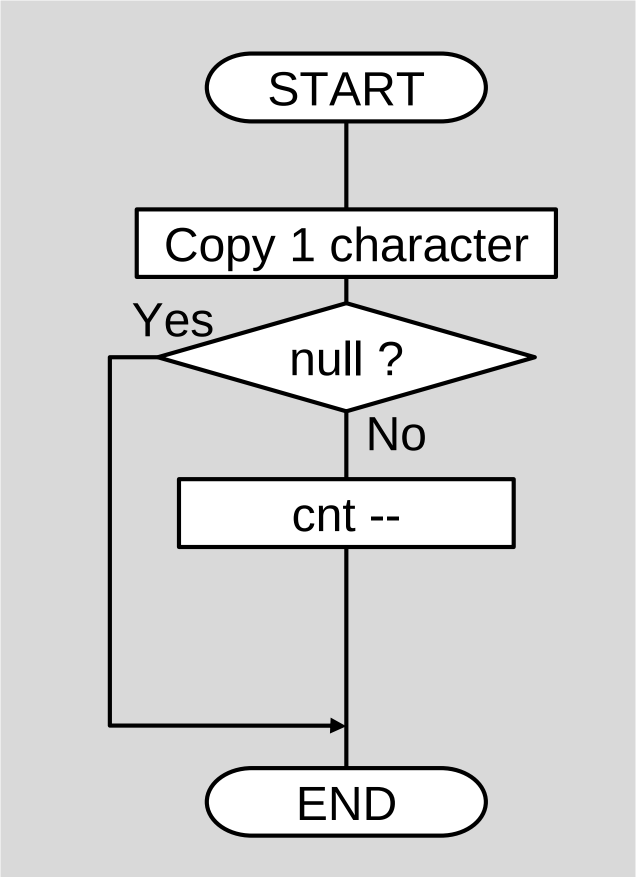
- Locate branch instruction at the bottom of loop
  “Skip branch → The loop is finished”



How to protect branch instruction

- Locate branch instruction at the bottom of loop
  “Skip branch → The loop is finished”

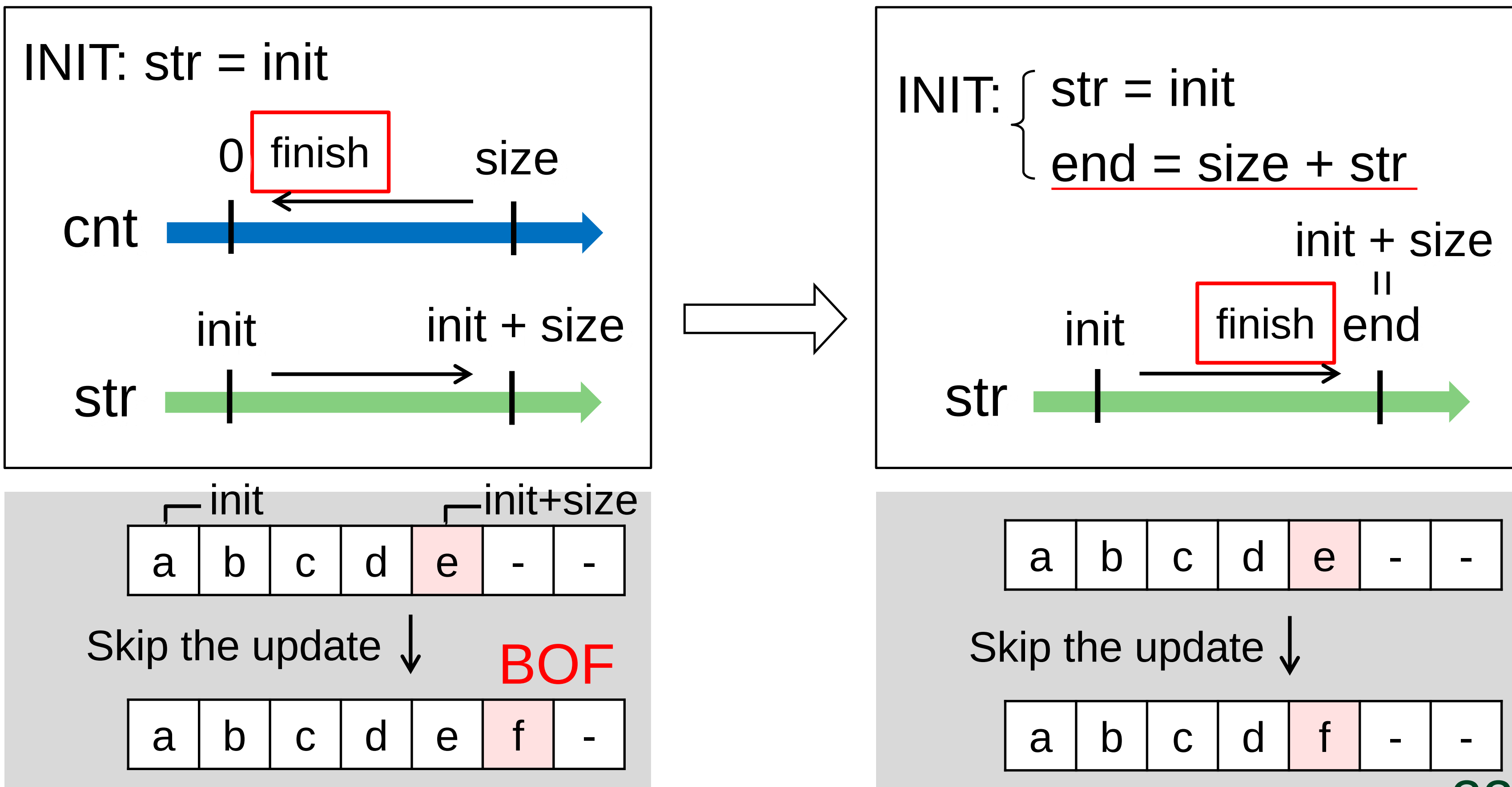




How to protect update instruction

- Associate loop counter with store address

  “Skip update → Data is stored into same address”



Application to AVR microcontroller

```
strncpy:
INIT:
      movw   r30, r22
      movw   r26, r24
LOOP:
      subi   r20, 0x01
      sbci   r21, 0x00
      brcs   END
      ld     r0, Z+
      st     X+, r0
      and    r0, r0
      brne   LOOP
      rjmp   Z_CMP
…
```

```
my_strncpy:
INIT:
      movw   r30, r22
      movw   r26, r24
      add    r20, r26
      adc    r21, r27
      rjmp   CMP
LOOP:
      ld     r0, Z+
      st     X, r0
      adiw   r26, 0x01
      and    r0, r0
      breq   Z_CMP
CMP:
      cp     r26, r20
      cpc    r27, r21
      brlo   LOOP
      ret
…
```

Protected instructions

- Update instructions
- Branch instruction

Security evaluation

- Assumptions of attackers
  1. Skip multiple and arbitrary instructions
  2. Use BOF
  3. All flags are reset when my_strncpy() is called

- Evaluation method
  - Examine all the possible instruction skips (2^{20} patterns)
  - Consider all the combinations of four instructions by above assumptions

```
my_strncpy: # 20 inst
INIT:
      movw    r30, r22
      movw    r26, r24
  (1) add     r20, r26
  (2) adc     r21, r27
      rjmp    CMP
LOOP:
      ld      r0, Z+
      st      X, r0
      adiw    r26, 0x01
      and     r0, r0
      breq    Z_CMP
CMP:
  (3) cp      r26, r20
  (4) cpc     r27, r21
      brlo    LOOP
      ret
 …
```

Examining skipping of **add**

size : original input size

size': input size after instruction skip

- *size'* > *size* ⇒ BOF happens

- $size' = end' - str'$

  str': initialized value of store address

- $size' - size = CF' * 0x100 - r26 \quad (0 \leq r26 \leq 0x100)$

  CF': carry flag when *my_strncpy()* is called

- If $CF' = 1$ then $size' > size$, and BOF happens

- But, according to the assumptions, all flags are reset and $CF' = 0$. So BOF **cannot** happen.

```
my_strncpy: # 20 inst
INIT:
    movw    r30, r22
    movw    r26, r24
    add     r20, r26
    adc     r21, r27
    rjmp    CMP
LOOP:
    ld      r0, Z+
    st      X, r0
    adiw    r26, 0x01
    and     r0, r0
    breq    Z_CMP
CMP:
    cp      r26, r20
    cpc     r27, r21
    brlo    LOOP
    ret
 …
```

Overheads by our countermeasure

Function name	Program memory [Byte]		Clock cycles	
	Total	Difference	Total	Difference
strncpy()	30	-	10 + 10n	-
my_strncpy()	40	+10	13 + 11n	+(3 + n)

* n: size, argument of strncpy()

Conclusion and future work

- Conclusion
  - Proposal of buffer overflow (BOF) attack with multiple fault injection
    - Invalidated typical software countermeasure against BOF attacks, and performed BOF attack
  - Proposal of software countermeasure, evaluated its overhead, and proved its validity

- Future work
  - Apply our attack to other microprocessors, such as ARM
  - Propose “systematic” proof of the countermeasure