Software Countermeasures for Control Flow Integrity of Smart Card C Codes

Jean-François Lalande
Karine Heydemann – Pascal Berthomé

Inria / Supélec (IRISA) – INSA CVL / Univ. Orléans (LIFO)
UPMC - (LIP6)

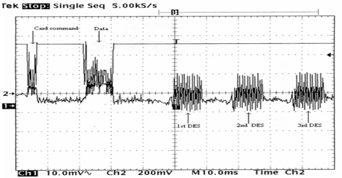
ESORICS 2014
September 7-11, Wroclaw, Poland

Introduction: ① smart card attacks

- Smart card are subject to physical attacks
- Security is of main importance for the card industry

Physical attacks:

- Means: laser beam, clock glitch, electromagnetic pulse, . . .
- Goal: disrupting execution of smartcard programs, producing a faulty execution

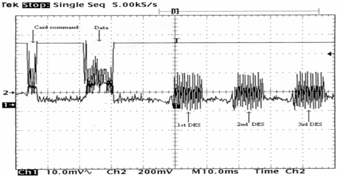


Introduction: (1) smart card attacks

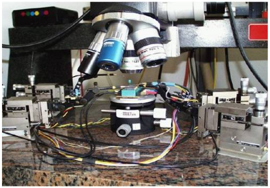
- Smart card are subject to physical attacks
- Security is of main importance for the card industry

Physical attacks:

- Means: laser beam, clock glitch, electromagnetic pulse, . . .
- Goal: disrupting execution of smartcard programs, producing a faulty execution



See this



Do this

Attack model

At **low level**, physical attacks can:

- induce a bit flip
- overwrite a bit/byte with controlled values
- overwrite a bit/byte with random bits

At **program level**, physical attacks can have different impacts:

- Disturb the value of some variables
- Modify the control flow by overwriting instructions when fetched:
  - Change a branch direction
  - Execute some NOPs
  - Execute an unconditional JMP

We focus on attacks that result in a jump, called a jump attack

Attack example

Let us consider such an authentication code:

```
uint user_tries = 0; // initialization of the number of tries for this session
uint max_tries = 3; // max number of tries
while (...) /* card life cycle: */
{
  incr_tries(user_tries);
  res = get_pin_from_terminal(); // receives 1234
  pin = read_secret_pin(); // read real pin: 0000
  if (compare(res, pin))
    { dec_tries(user_tries);
      do_stuff(); }
  if (user_tries >= max_tries)
    { killcard(); }
}
```

Simplified authentication code with pin check

Attack example

Let us consider such an authentication code:

```
uint user_tries = 0; // initialization of the number of tries for this session
uint max_tries = 3; // max number of tries
while (...) /* card life cycle: */
{
  incr_tries(user_tries);
  res = get_pin_from_terminal(); // receives 1234
  pin = read_secret_pin(); // read real pin: 0000
  if (compare(res, pin))  ⇒ NOP ... NOP
    { dec_tries(user_tries);
      do_stuff(); }
  if (user_tries >= max_tries)
    { killcard(); }
}
```

Simplified authentication code with pin check

Security problems and contributions

Several questions appear:

- How to deal with low level attacks when working at source code level?

Use a high level model of attacks

- How to identify harmful attacks?

Simulate attacks and distinguish weaknesses

- How to implement countermeasures?

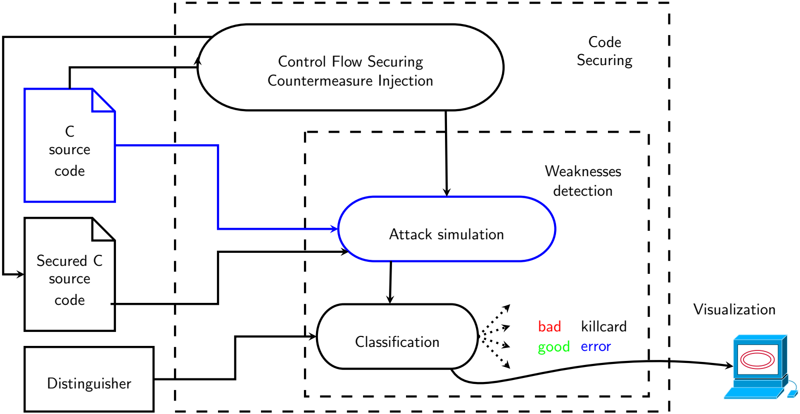
Protect code at source level using counters

- Are the proposed countermeasures effective?

Study formally and experimentally their effectiveness

Outline

② Weaknesses detection
⋆ Attack simulation ⋆ Distinguisher ⋆ Analysis result



Simulation of jump attacks

```
void aes_addRoundKey_cpy(uint8_t *buf, uint8_t *key, uint8_t *cpk)
{
    register uint8_t i = 16;

    while (i--)
    {
      buf[i] ^= key[i];
      cpk[i] = key[i];
      cpk[16+i] = key[16 + i];
    }
    ;
} /* aes_addRoundKey_cpy */
```

Function of an implementation of AES

Simulation by insertion of jump attack

Simulation of jump attacks

```
void aes_addRoundKey_cpy(uint8_t *buf, uint8_t *key, uint8_t *cpk)
{
    register uint8_t i = 16;
 goto dest;
    while (i--)
    {
  dest:buf[i] ^= key[i];
      cpk[i] = key[i];
      cpk[16+i] = key[16 + i];
    }
    ;
} /* aes_addRoundKey_cpy */
```

Function of an implementation of AES

Simulation by insertion of jump attack

Simulation of jump attacks

```
void aes_addRoundKey_cpy(uint8_t *buf, uint8_t *key, uint8_t *cpk)
{
    register uint8_t i = 16;
 goto dest;
    while (i--)
    {
      buf[i] ^= key[i];
  dest:cpk[i] = key[i];
      cpk[16+i] = key[16 + i];
    }
    ;
} /* aes_addRoundKey_cpy */
```

Function of an implementation of AES

Simulation by insertion of jump attack

Simulation of jump attacks

```
void aes_addRoundKey_cpy(uint8_t *buf, uint8_t *key, uint8_t *cpk)
{
    register uint8_t i = 16;
 goto dest;
    while (i--)
    {
      buf[i] ^= key[i];
      cpk[i] = key[i];
  dest:cpk[16+i] = key[16 + i];
    }
    ;
} /* aes_addRoundKey_cpy */
```

Function of an implementation of AES

Simulation by insertion of jump attack

Simulation of jump attacks

```
void aes_addRoundKey_cpy(uint8_t *buf, uint8_t *key, uint8_t *cpk)
{
    register uint8_t i = 16;
 goto dest;
    while (i--)
    {
      buf[i] ^= key[i];
      cpk[i] = key[i];
      cpk[16+i] = key[16 + i];
dest:}
    ;
} /* aes_addRoundKey_cpy */
```

Function of an implementation of AES

Simulation by insertion of jump attack

Simulation of jump attacks

```
void aes_addRoundKey_cpy(uint8_t *buf, uint8_t *key, uint8_t *cpk)
{
    register uint8_t i = 16;
 goto dest;
    while (i--)
    {
      buf[i] ^= key[i];
      cpk[i] = key[i];
      cpk[16+i] = key[16 + i];
    }
dest:;
} /* aes_addRoundKey_cpy */
```

Function of an implementation of AES

Simulation by insertion of jump attack

Simulation of jump attacks

```
void aes_addRoundKey_cpy(uint8_t *buf, uint8_t *key, uint8_t *cpk)
{
    register uint8_t i = 16;
 dest:
    while (i--)
    {
      buf[i] ^= key[i];
      cpk[i] = key[i];
      cpk[16+i] = key[16 + i];
    }
    ;  goto dest;
} /* aes_addRoundKey_cpy */
```

Function of an implementation of AES

Simulation by insertion of jump attack

Simulation of jump attacks

```
void aes_addRoundKey_cpy(uint8_t *buf, uint8_t *key, uint8_t *cpk)
{
    register uint8_t i = 16;

    while (i--)
    {
  dest:buf[i] ^= key[i];
        cpk[i] = key[i];
        cpk[16+i] = key[16 + i];
    }
    ;  goto dest;
} /* aes_addRoundKey_cpy */
```

Function of an implementation of AES

Simulation by insertion of jump attack

Simulation of jump attacks

```
void aes_addRoundKey_cpy(uint8_t *buf, uint8_t *key, uint8_t *cpk)
{
    register uint8_t i = 16;

    while (i--)
    {
      buf[i] ^= key[i];
  dest:cpk[i] = key[i];
      cpk[16+i] = key[16 + i];
    }
    ;  goto dest;
} /* aes_addRoundKey_cpy */
```

Function of an implementation of AES

Simulation by insertion of jump attack

Simulation of jump attacks

```
void aes_addRoundKey_cpy(uint8_t *buf, uint8_t *key, uint8_t *cpk)
{
    register uint8_t i = 16;

    while (i--)
    {
      buf[i] ^= key[i];
      cpk[i] = key[i];
  dest:cpk[16+i] = key[16 + i];
    }
    ;  goto dest;
} /* aes_addRoundKey_cpy */
```

Function of an implementation of AES

Simulation by insertion of jump attack

Simulation of jump attacks

```
void aes_addRoundKey_cpy(uint8_t *buf, uint8_t *key, uint8_t *cpk)
{
    register uint8_t i = 16;

    while (i--)
    {
      buf[i] ^= key[i];
      cpk[i] = key[i];
      cpk[16+i] = key[16 + i];
dest:}
    ;  goto dest;
} /* aes_addRoundKey_cpy */
```

Function of an implementation of AES

Simulation by insertion of jump attack

Simulation of jump attacks

```
void aes_addRoundKey_cpy(uint8_t *buf, uint8_t *key, uint8_t *cpk)
{
    register uint8_t i = 16;
 dest:
    while (i--)
    {
      buf[i] ^= key[i];
      cpk[i] = key[i];  goto dest; // 16 ≠ triggering times
      cpk[16+i] = key[16 + i];
    }
    ;
} /* aes_addRoundKey_cpy */
```

Function of an implementation of AES

Full coverage of attacks simulation by using gcov information

Simulation of jump attacks

```
void aes_addRoundKey_cpy(uint8_t *buf, uint8_t *key, uint8_t *cpk)
{
    register uint8_t i = 16;
 dest:
    while (i--)
    {
      buf[i] ^= key[i];
      cpk[i] = key[i];  if (trigger time) goto dest; // 16 ≠ triggerring times
      cpk[16+i] = key[16 + i];
    }
    ;
} /* aes_addRoundKey_cpy */
```

Function of an implementation of AES

Full coverage of attacks simulation by using gcov information

Simulation of jump attacks

```
void aes_addRoundKey_cpy(uint8_t *buf, uint8_t *key, uint8_t *cpk)
{
    register uint8_t i = 16;

    while (i--)
    {
  dest:buf[i] ^= key[i];
      cpk[i] = key[i];  if (trigger time) goto dest; // 16 ≠ triggerring times
      cpk[16+i] = key[16 + i];
    }
    ;
} /* aes_addRoundKey_cpy */
```

Function of an implementation of AES

Full coverage of attacks simulation by using gcov information

Simulation of jump attacks

```c
void aes_addRoundKey_cpy(uint8_t *buf, uint8_t *key, uint8_t *cpk)
{
    register uint8_t i = 16;

    while (i--)
    {
      buf[i] ^= key[i];
  dest:cpk[i] = key[i];  if (trigger time) goto dest; // 16 ≠ triggerring times
      cpk[16+i] = key[16 + i];
    }
    ;
} /* aes_addRoundKey_cpy */
```

Function of an implementation of AES

Full coverage of attacks simulation by using gcov information

Simulation of jump attacks

```
void aes_addRoundKey_cpy(uint8_t *buf, uint8_t *key, uint8_t *cpk)
{
    register uint8_t i = 16;

    while (i--)
    {
      buf[i] ^= key[i];
      cpk[i] = key[i];  if (trigger time) goto dest; // 16 ≠ triggerring times
      cpk[16+i] = key[16 + i];
dest:}
    ;
} /* aes_addRoundKey_cpy */
```

Function of an implementation of AES

Full coverage of attacks simulation by using gcov information

Simulation of jump attacks

```
void aes_addRoundKey_cpy(uint8_t *buf, uint8_t *key, uint8_t *cpk)
{
    register uint8_t i = 16;

    while (i--)
    {
      buf[i] ^= key[i];
      cpk[i] = key[i];  if (trigger time) goto dest; // 16 ≠ triggerring times
      cpk[16+i] = key[16 + i];
    }
dest:;
} /* aes_addRoundKey_cpy */
```

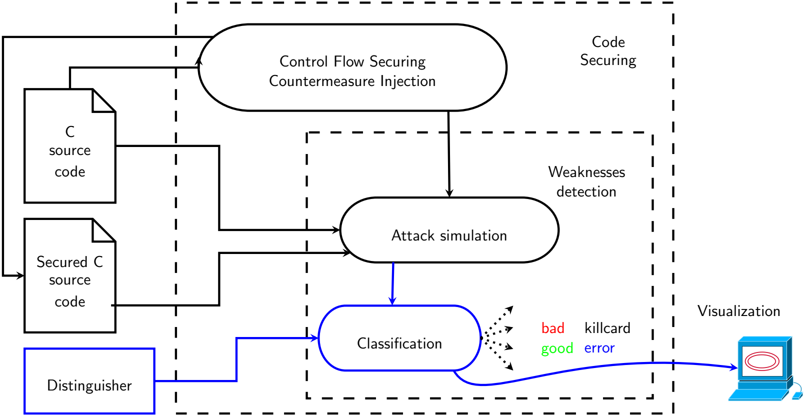
Function of an implementation of AES

Full coverage of attacks simulation by using gcov information

Harmful and harmless attacks classification

How to evaluate the effect of (simulated) attacks?

- define a **functional scenario** (with fixed inputs/outputs):
- be able to **distinguish** unexpected from expected outputs



Attacks classification

Considered scenario

Encryption of a fixed input by AES (Levin 07), SHA and Blowfish (Guthaus et al. 01)

Attacks classification

Considered scenario

Encryption of a fixed input by AES (Levin 07), SHA and Blowfish (Guthaus et al. 01)

Distinguisher classes (harmful/harmless):

- **bad**: during execution a benefit has been obtained by the attacker;
  - **bad j>1**: (*jumpsize* $\geq$ 2 lines) the encryption output is wrong;
  - **bad j=1**: (*jumpsize* $=$ 1 line) the encryption output is wrong;

Attacks classification

Considered scenario

Encryption of a fixed input by AES (Levin 07), SHA and Blowfish (Guthaus et al. 01)

Distinguisher classes (harmful/harmless):

- **bad**: during execution a benefit has been obtained by the attacker;
  - **bad j>1**: (*jumpsize* ≥ 2 lines) the encryption output is wrong;
  - **bad j=1**: (*jumpsize* $= 1$ line) the encryption output is wrong;
- **good**: output is unchanged

Attacks classification

Considered scenario

Encryption of a fixed input by AES (Levin 07), SHA and Blowfish (Guthaus et al. 01)

Distinguisher classes (harmful/harmless):

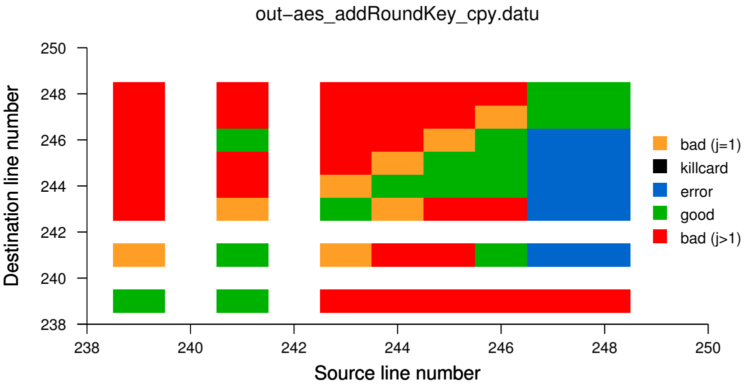
- **bad**: during execution a benefit has been obtained by the attacker;
  - **bad j>1**: (*jumpsize* $\geq$ 2 lines) the encryption output is wrong;
  - **bad j=1**: (*jumpsize* $=$ 1 line) the encryption output is wrong;
- **good**: output is unchanged
- **error** or **timeout**: error, crash, infinite loop;
- **killcard**: attack detected: the card is turned out of service!

Weaknesses detection results

	bad j > 1	bad j = 1	good	error	total
C JUMP ATTACKS	Attacking all functions at C level for all transient rounds				
AES	7786 29%	1104 4.2%	17372 65%	108 0.4%	26370 100%
SHA	32818 75%	1528 3.5%	8516 19%	412 1.0%	43274 100%
Blowfish	70086 32%	3550 1.7%	134360 62%	5725 2.7%	213721 100%

- **bad j>1**: (*jumpsize* $\geq$ 2 lines) the encryption output is wrong;
- **bad j=1**: (*jumpsize* $=$ 1 line) the encryption output is wrong;

Weaknesses visualization

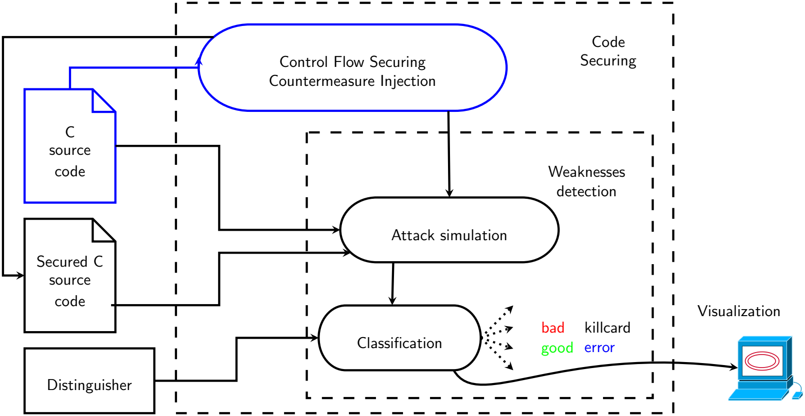


Visualization of weaknesses for aes_addRoundKey_cpy

Outline

3 Code securing

⋆ Securing control flow constructs ⋆ Verifying countermeasures robustness ⋆ Experimental results



Goals

Code securing techniques for **C**ontrol **F**low **I**ntegrity often rely on:

- Modified assembly codes (Abadi et al. 05)
- Modified JVM (Iguchi-cartigny et al. 11, Lackner et al. 13)
- Signature techniques of each basic block (Oh et al. 02, Nicolescu et al. 03)

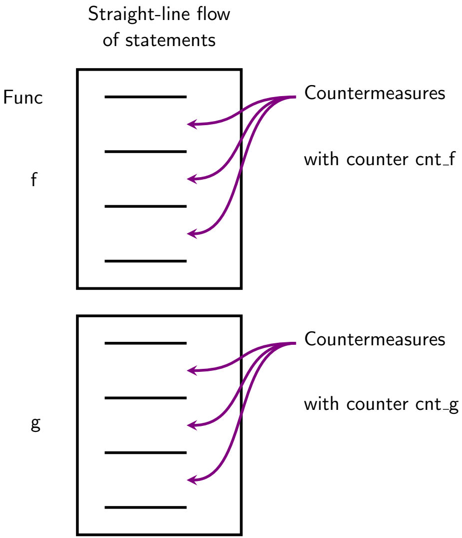
We aim at keeping the assembly code intact:

- A certified compiler enable to certify the secured program
- ⇒ CFI countermeasures to be compiled by a certified compiler

Checks often performed at entry/exit of basic blocks:

- CFI countermeasures should also check the flow inside basic blocks

Securing principle



Countermeasures

- 1 counter by function
- between two statements

Check of counter values

cnt = (cnt == val+N ?
cnt +1 : killcard());

Securing details

Source code

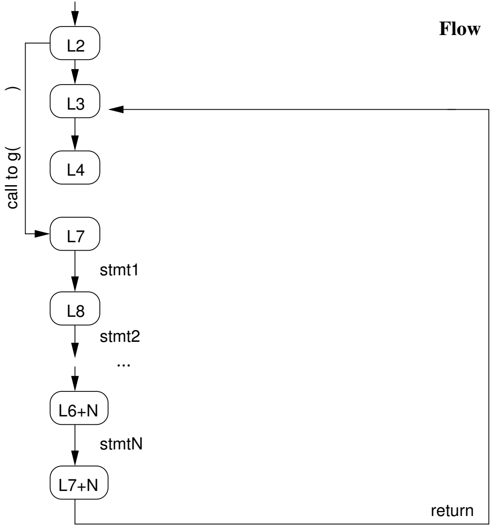
```
        void f(){
L1:

L2:     g(          );
L3:
L4:     }
        void g(               ){

L7:     stmt1;

L8:     stmt2;

              ...

L6+N:   stmtN;

L7+N:   return;
       }
```

Securing details

Source code

```
      void f(){
L1:

L2:   g(        );
L3:
L4:   }
      void g(              ){

L7:   stmt1;

L8:   stmt2;

            ...

L6+N: stmtN;

L7+N: return;
      }
```

Flow



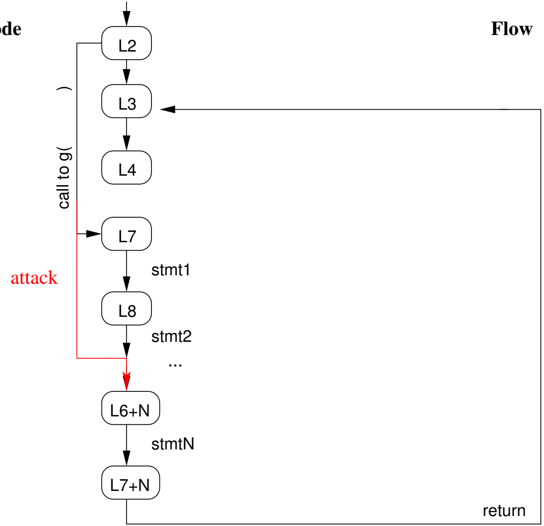
Securing details

Source code

```
      void f(){
L1:

L2:   g(          );
L3:
L4:   }
      void g(               ){

L7:   stmt1;

L8:   stmt2;

            ...

L6+N: stmtN;

L7+N: return;
      }
```

attack

Flow

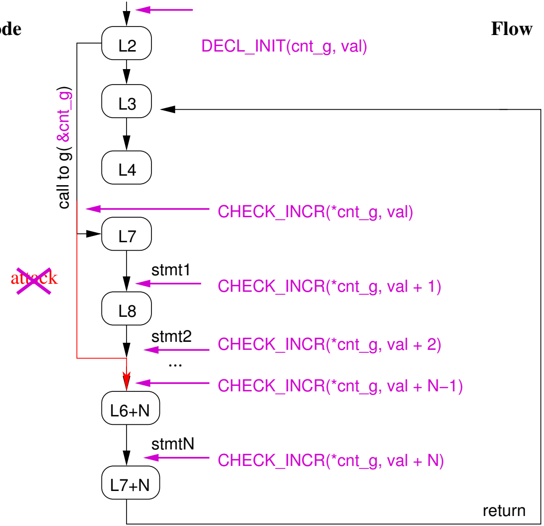


Securing details

Source code

```
        void f(){
L1:
        DECL_INIT(cnt_g, val)
L2:     g(&cnt_g);
L3:
L4:     }
        void g(            ){
        CHECK_INCR(*cnt_g, val)
L7:     stmt1;
        CHECK_INCR(*cnt_g, val + 1)
L8:     stmt2;          attack
        CHECK_INCR(*cnt_g, val + 2)
        ...
        CHECK_INCR(*cnt_g, val + N–1)
L6+N:   stmtN;
        CHECK_INCR(*cnt_g, val + N)
L7+N:   return;
        }
```

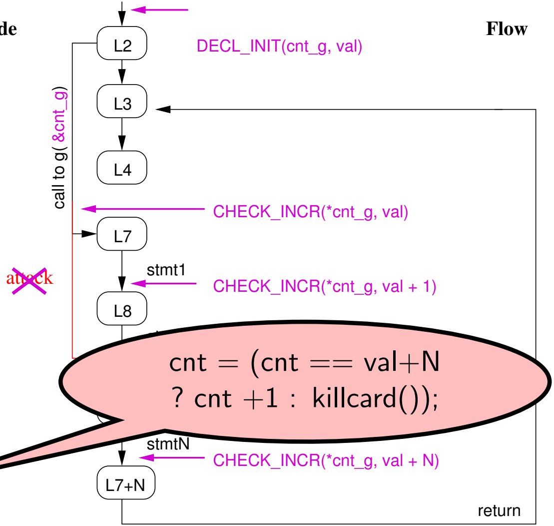
Flow



Securing details

Source code

```
    void f(){
L1:
    DECL_INIT(cnt_g, val)

L2: g(&cnt_g);
L3:
L4: }
    void g(          ){
    CHECK_INCR(*cnt_g, val)
L7: stmt1;
    CHECK_INCR(*cnt_g, val + 1)
L8: stmt2;          attack
    CHECK_INCR(*cnt_g, val + 2)
        ...
    CHECK_INCR(*cnt_g, val + N−1)
L6+N: stmtN;
    CHECK_INCR(*cnt_g, val + N)
L7+N: return;
    }
```



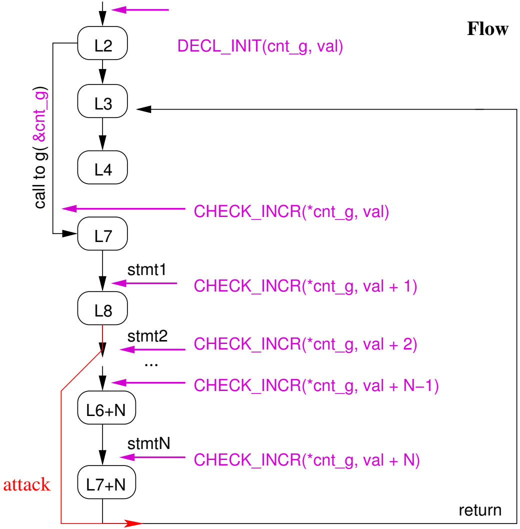
cnt = (cnt == val+N
? cnt +1 : killcard());

Securing details

Source code

```
      void f(){
L1:
      DECL_INIT(cnt_g, val)
L2:   g(&cnt_g);
L3:
L4:   }
      void g(            ){
      CHECK_INCR(*cnt_g, val)
L7:   stmt1;
      CHECK_INCR(*cnt_g, val + 1)
L8:   stmt2;
      CHECK_INCR(*cnt_g, val + 2)
        ...
      CHECK_INCR(*cnt_g, val + N–1)
L6+N: stmtN;     attack
      CHECK_INCR(*cnt_g, val + N)
L7+N: return;
      }
```

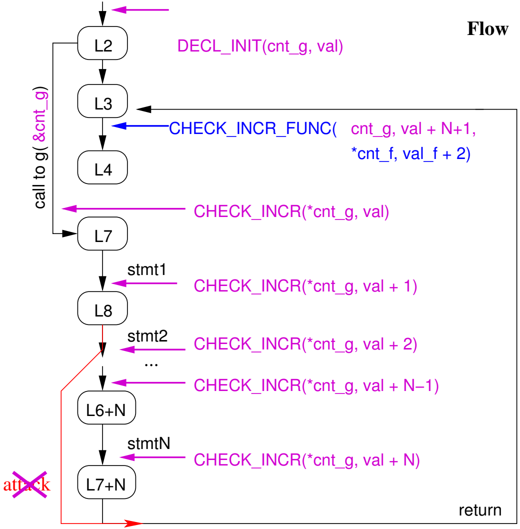
Flow



Securing details

Source code

```
      void f(){
L1:
      DECL_INIT(cnt_g, val)

L2:   g(&cnt_g);
L3:   CHECK_INCR_FUNC( cnt_g, val + N+1,
L4:   }                 *cnt_f, val_f + 2)
      void g(          ){
      CHECK_INCR(*cnt_g, val)

L7:   stmt1;

      CHECK_INCR(*cnt_g, val + 1)

L8:   stmt2;

      CHECK_INCR(*cnt_g, val + 2)
          ...
      CHECK_INCR(*cnt_g, val + N-1)
L6+N: stmtN;       attack

      CHECK_INCR(*cnt_g, val + N)
L7+N: return;
      }
```

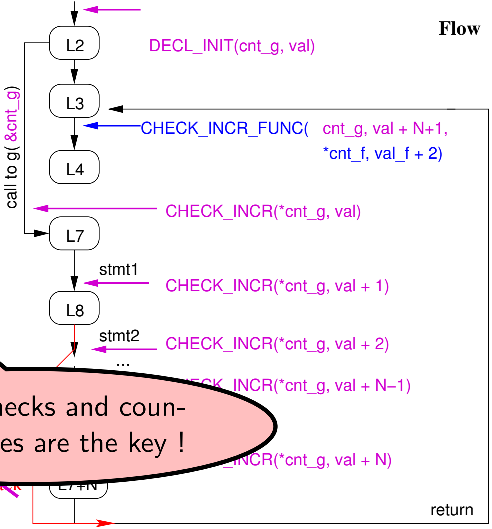


Securing details

Source code

```
      void f(){
L1:
      DECL_INIT(cnt_g, val)

L2:   g(&cnt_g);
L3:   CHECK_INCR_FUNC( cnt_g, val + N+1,
L4:   }                     *cnt_f, val_f + 2)
      void g(              ){
      CHECK_INCR(*cnt_g, val)

L7:   stmt1;
      CHECK_INCR(*cnt_g, val + 1)

L8:   stmt2;
      CHECK_INCR(*cnt_g, val + 2)
      ...
      CHECK_INCR(*cnt
L6+N: stmtN;       att
      CHECK_INCR(*cnt_g, v
L7+N: return;
}
```

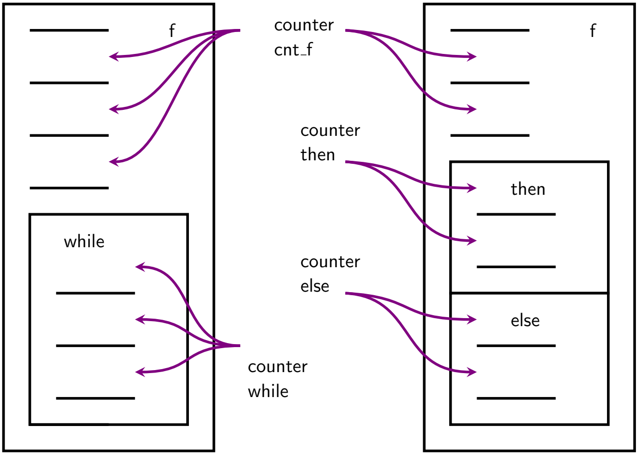
Flow



Nesting checks and counters updates are the key !

Securing loops and conditional constructs

Countermeasures also designed for **while**/**if** constructs



Countermeasure robustness?

Are these countermeasures effective for all possible jump attacks?

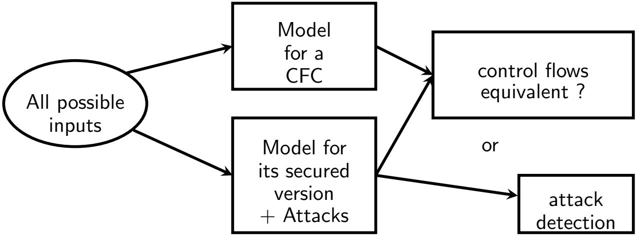
- of course not, for a jump size equal to 1 C line!
- what about attacks with jump size $\geq$ 2 C lines?

Countermeasure robustness?

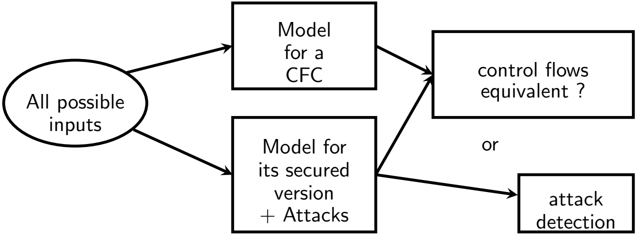
Are these countermeasures effective for all possible jump attacks?

- of course not, for a jump size equal to 1 C line!
- what about attacks with jump size $\geq$ 2 C lines?

We model a **C**ontrol **F**low **C**onstruct (CFC) with a transition system to verify countermeasure robustness and flow correctness



Formal verification of robustness



Our securing scheme for **if**, **loops** and **sequential** control flow constructs verify:

- any jump attack of more than 2 C lines is detected
- or the control flow is correct

Verification performed with **VIS** model checker

Experimental results I

Jump attacks simulated in the secured source code

C JUMP ATTACKS	bad j > 1	bad j = 1	good	killcard	error	total
	Attacking all functions at C level for all transient rounds					
AES	29%	4.2%	65%		0.4%	26370
AES + CM	0%	0.2%	5.3%	94%	0.0%	337516
SHA	75%	3.5%	19%		1.0%	43274
SHA + CM	0%	0.3%	1.2%	98%	0.1%	427690
Blowfish	32%	1.7%	62%		2.7%	213721
Blowfish + CM	0%	0.2%	23%	75%	0.4%	1400355

Jump attacks simulated at C level

100% of harmfull attacks jumping more than 2 C lines are captured

Experimental results II

- Simulation of jump attacks at assembly level
- ASM attacks injected on the fly using an ARM simulator

	bad j > 1	bad j = 1	good	killcard	error	total
ASM JUMP ATT.	Attacking the aes_encrypt function at ASM level for the first transient round					
aes_encrypt	82.8%	1.9%	9.4%		5.9%	1892
aes_encrypt + CM	0.2%	~0%	20.2%	78.4%	0.7%	305255

Jump attacks simulated at ASM level

- Reduction: 60% of harmfull attack are detected
- Remaining attacks are harder to perform (82.8% ⇒ 0.2%)

Experimental results III

- Simulation of function call attacks
- ASM attacks injected on the fly using an ARM simulator

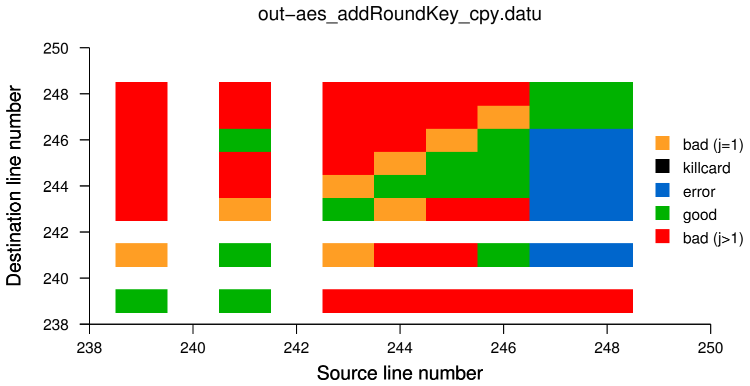
	bad j > 1	bad j = 1	good	killcard	error	total
ASM CALL ATT.	Attacking all function calls at ASM level for the first transient round					
AES	59.3%		33.1%		5%	420
AES + CM	0%		5%	94.8%	0.2%	420
SHA	48.7%		18%		33.3%	72
SHA + CM	0%		11.1%	84.7%	4.2%	72
Blowfish	21.4%		42.9%		35.7%	42
Blowfish + CM	0%		42.9%	40.5%	16.6%	42

Jump attacks simulated at ASM level

Experimental results IV

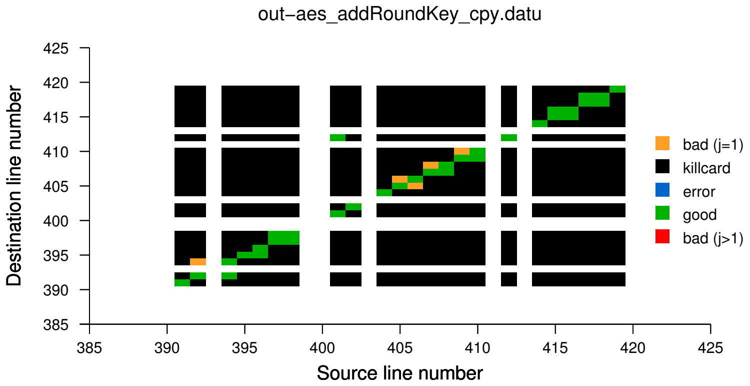
- 100% of harmfull attacks are captured

Weaknesses visualization



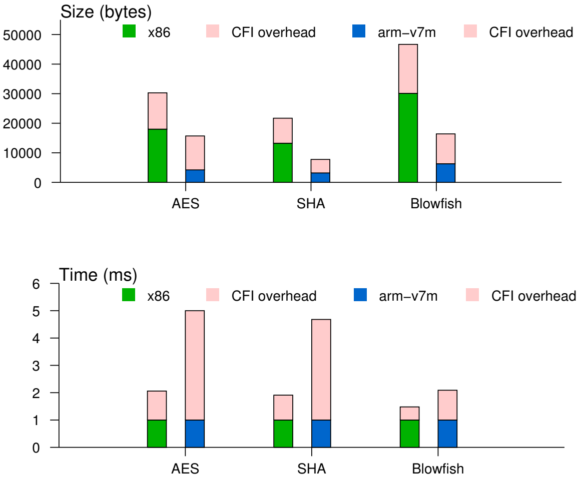
Visualization of weaknesses for aes_addRoundKey_cpy

Weaknesses visualization with CFI



Visualization of weaknesses for the secured version

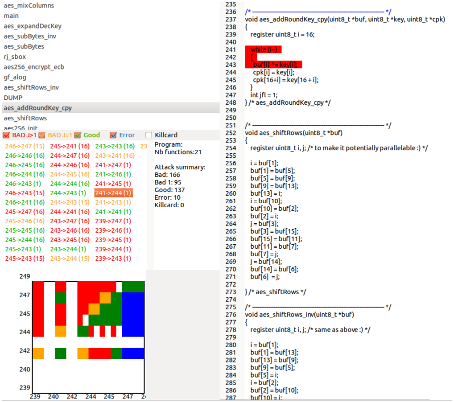
Securing code overheads - x86 and arm-v7m



Overhead for 1ms of computation

Demo

Demo: graphical tool for navigating into attacks !

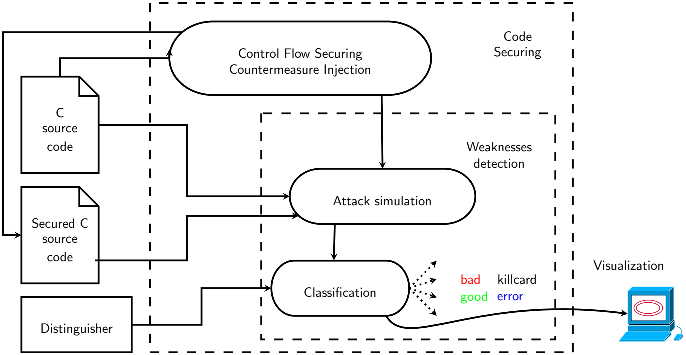


http://dai.ly/x205n3x

Conclusion

Software coutermeasures for control flow integrity

- Software-only effective countermeasures
- Protection for jump attacks than more than 1 C statement



Future work

New problems remain to be addressed

- Reduce overhead!
- Deal with jump attack of size one

And new challenges

- Is this suitable for javacard apps?
- Can we design software countermeasures for attacks impacting variable values?

Thank you!

Thank you!

...

(Diode Laser Station from Riscure)

Thank you!

(Diode Laser Station from Riscure)