Contre-mesures logicielles contre les fautes induisant des sauts

Jean-François Lalande
Karine Heydemann – Pascal Berthomé

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

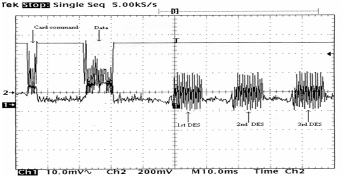
Workshop SERTIF
11 octobre 2016

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

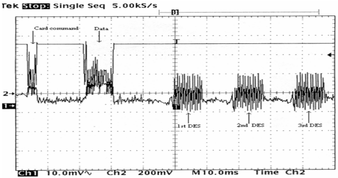


Introduction: (1) smart card attacks

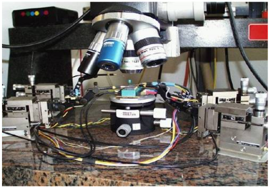
- Smart card are subject to physical attacks
- Security is of main importance for the card industry

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

- 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
⇒ Thèse X. Kauffmann-Tourkestansky

- How to implement countermeasures?

Protect code at source level using counters

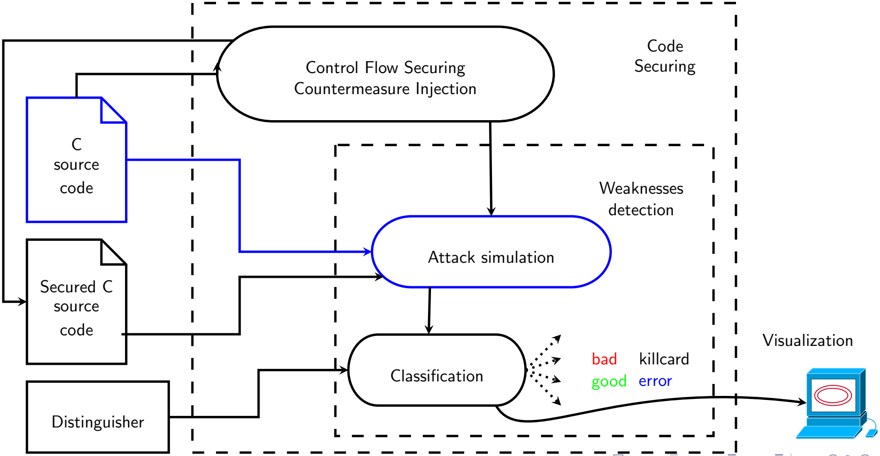
- Are the proposed countermeasures effective?

Study formally and experimentally their effectiveness

Outline

② Weaknesses detection

@JLL: l'outil s'appelle **cfi-c**: `http://cfi-c.gforge.inria.fr/`



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

```
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 */
```

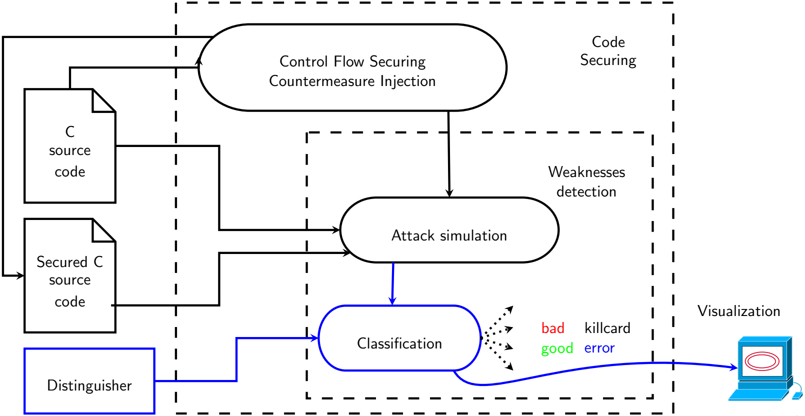
Function of an implementation of AES

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

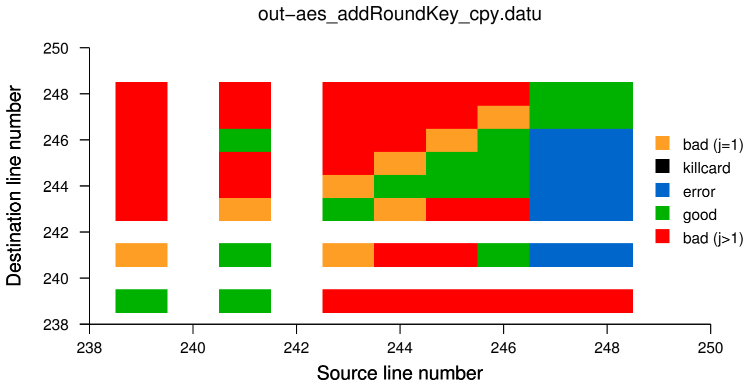
- **bad** (Wrong Answer):
  - **bad j>1**: (*jumpsize* ≥ 2 lines) the encryption output is wrong;
  - **bad j=1**: (*jumpsize* $= 1$ line) the encryption output is wrong;
- **good** (Effect Less): output is unchanged
- **error** or **timeout**: error, crash, infinite loop;
- **killcard** (Detection): attack detected

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%

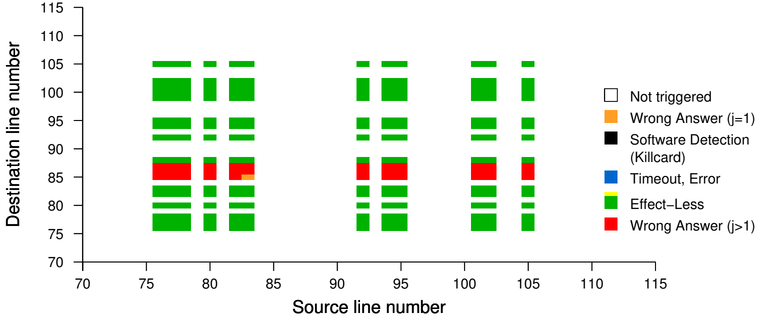
- **bad j>1**: (*jumpsize* $\geq$ 2 lines) the encryption output is wrong;
- **bad j=1**: (*jumpsize* $=$ 1 line) the encryption output is wrong;

Weaknesses visualization: AES



Visualization of weaknesses for aes_addRoundKey_cpy

Weaknesses visualization: FISSC (Dureuil et al. 16)



Visualization of verifyPIN_1 (FISSC - Dureuil et al. 16)

BOOL verifyPIN_1() du benchmark FISSC

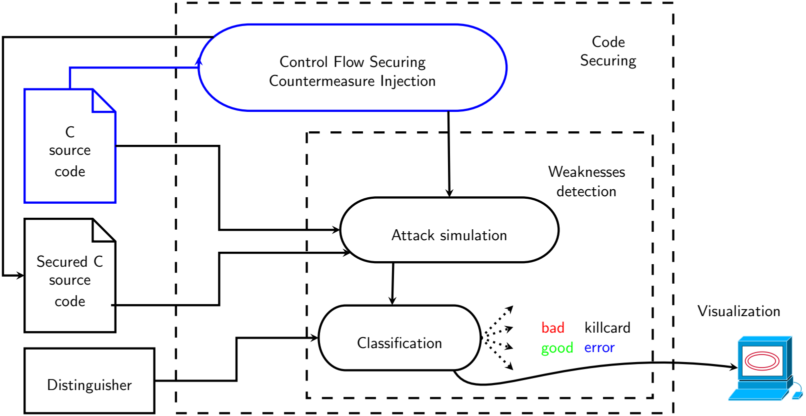
```
if(g_ptc > 0)
{
  comp = byteArrayCompare(g_userPin, g_cardPin, PIN_SIZE);
  if(comp == BOOL_TRUE)
  {
    g_ptc = 3;
    g_authenticated = BOOL_TRUE; // Authentication();
    printf("auth\n");
    ret = BOOL_TRUE;
  }
```

BOOL verifyPIN_1()

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)

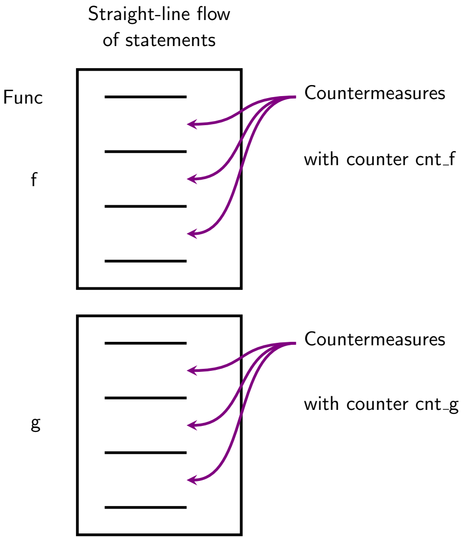
We aim at keeping the assembly code intact:

- A certified compiler enable to certify the secured program
- $\Rightarrow$ 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

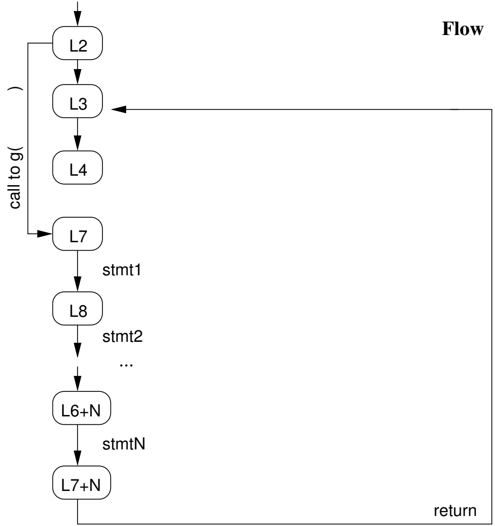
```
       void f(){
L1:

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



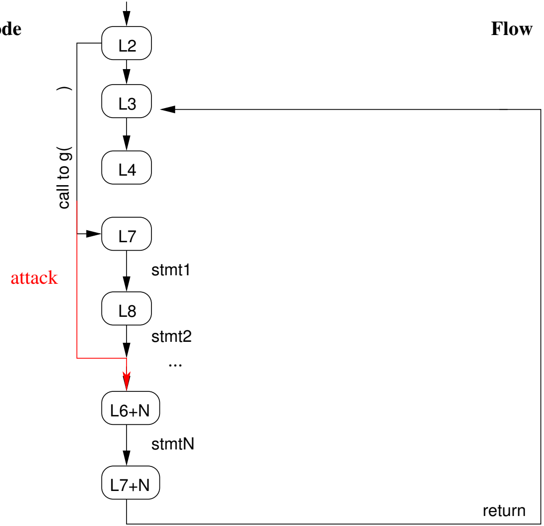
Securing details

Source code

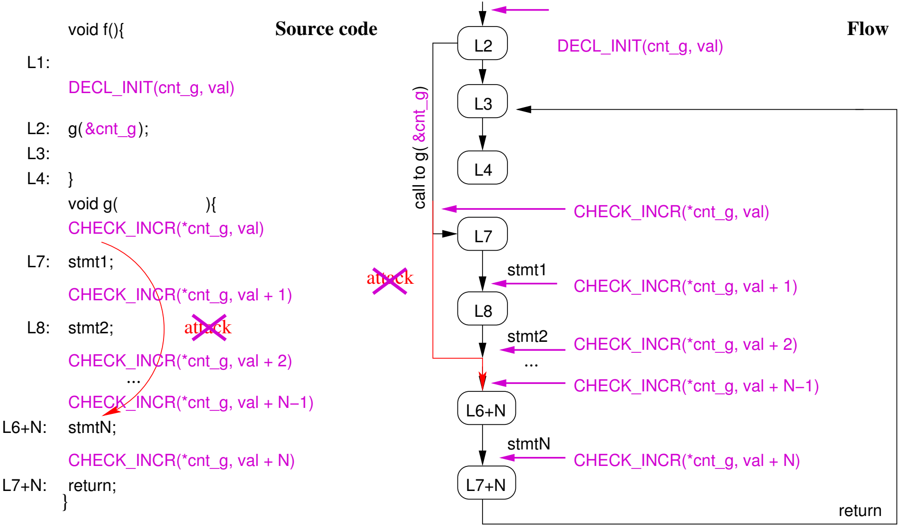
```
      void f(){
L1:
L2:   g(        );
L3:
L4:   }
      void g(             ){
L7:   stmt1;
L8:   stmt2;
      ...
L6+N: stmtN;
L7+N: return;
      }
```

attack

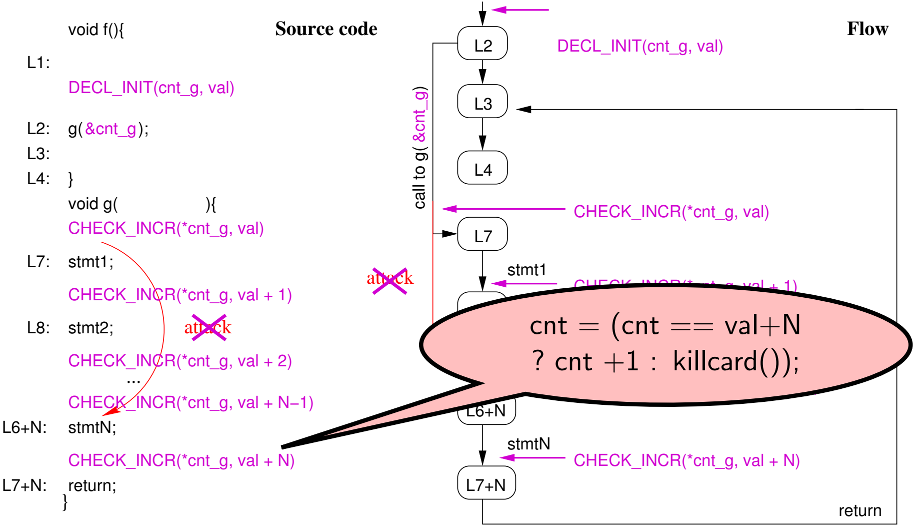
Flow



Securing details



Securing details

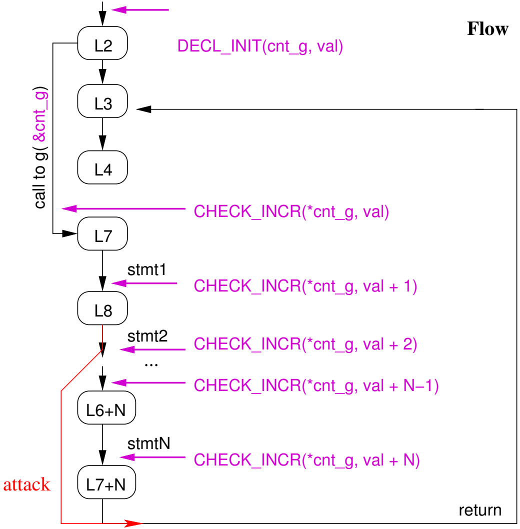


Securing details

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;
        }
```

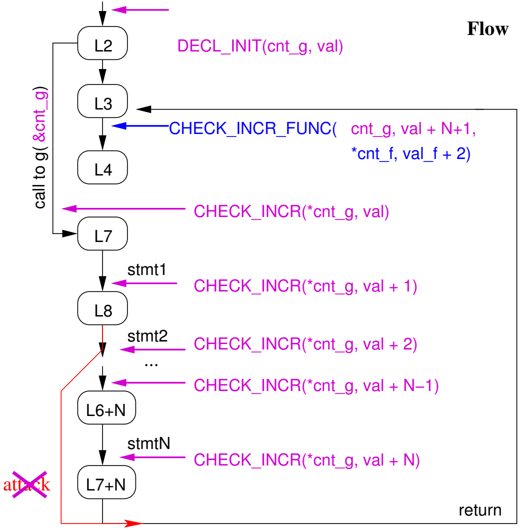
Flow



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;
      }
```

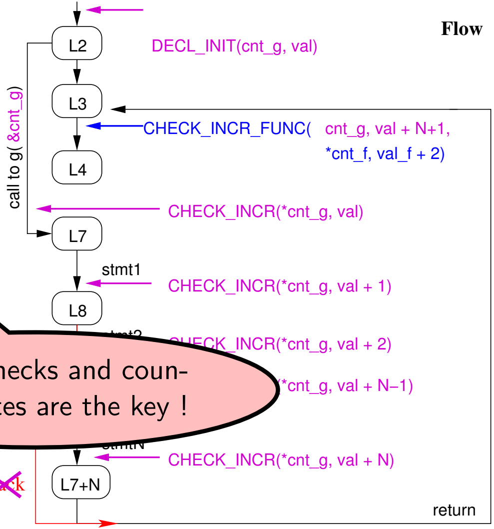


Securing details

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_
      ...
      CHECK_INCR(*c
L6+N: stmtN;           attack
      CHECK_INCR(*cnt_g, val + N)
L7+N: return;
      }
```

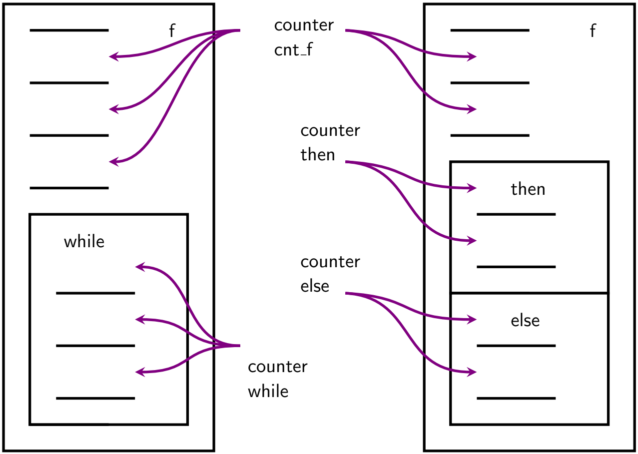
Flow



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?

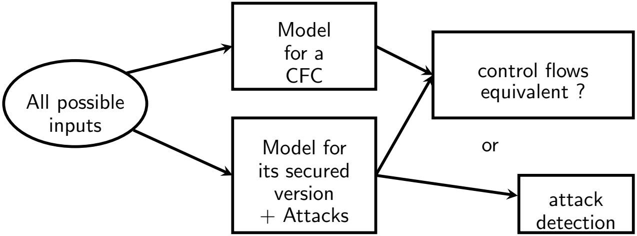
- of course not, for a jump size equal to 1 C line!
- what about attacks with jump size ≥ 2 C lines?

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?

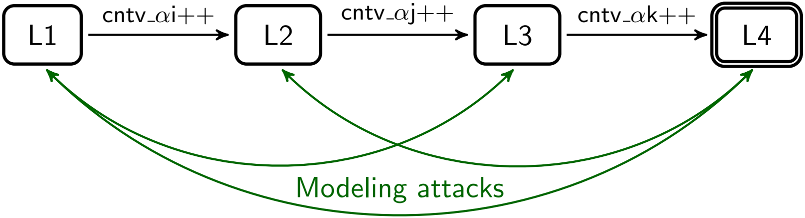
We model a **C**ontrol **F**low **C**onstruct (CFC) with a transition system to verify countermeasure robustness and flow correctness



Modeling jump attacks

Two models:

- M(c): model for initial control-flow construct
- CM(c): model including countermeasures and attacks



Robustness verification

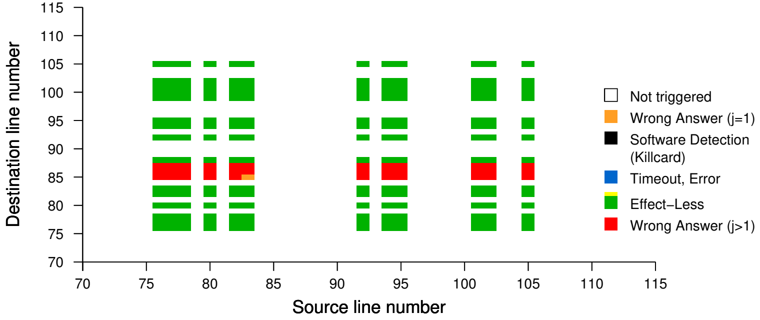
`M(c)` and `CM(c)` are proved to be sound by **VIS** (model checker)

In particular:

- statement counters are equal in `M(c)` and `CM(c)` (final states)
- $1 \geq$ `cntv_`α`i` $\geq$ `cntv_`α`(i+1)` ≥ 0 i.e.
  statement $i+1$ is performed after statement i and only once

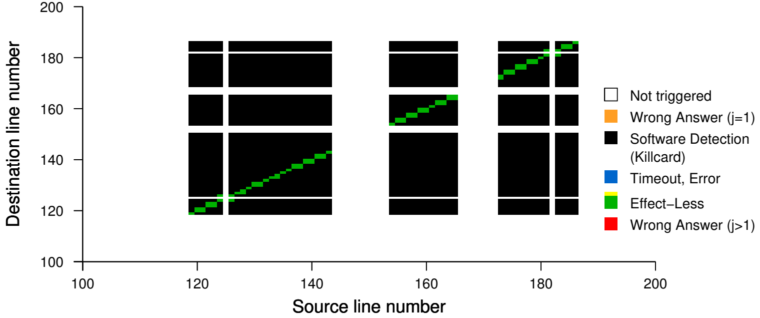
Models have also been designed for verifying
our securing scheme for **if** and **while** constructs

Weaknesses visualization: FISSC



Visualization of VerifyPIN_1 (FISSC)

Weaknesses visualization: Secured FISSC



Visualization of verifyPIN_1 + CM (secured)

Available in FISSC !

Experimental results I

Jump attacks simulated in the secured source code

	bad j > 1	bad j = 1	good	killcard	error	total
C JUMP ATTACKS	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

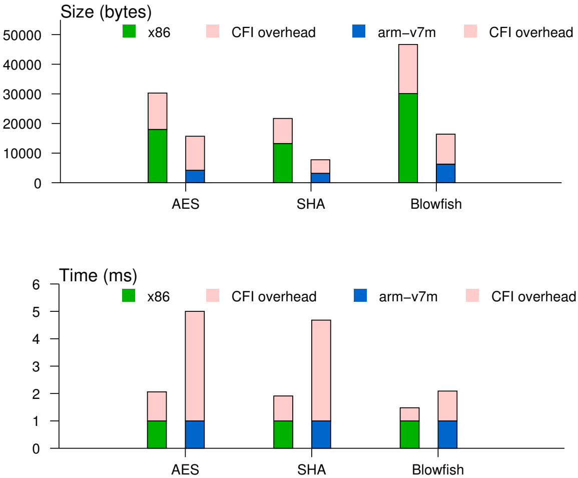
- Simulation of jump attacks at assembly level
- ASM attacks injected on the fly using an ARM simulator

	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%)

Securing code overheads - x86 and arm-v7m



Overhead for 1ms of computation

Conclusion

Software coutermeasures for control flow integrity

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

New challenges

- Deal with jump attack of size one
- Is this suitable for javacard apps?
- Can we design software countermeasures for attacks impacting variable values?

Thank you!

(Diode Laser Station from Riscure)

$\Rightarrow$

"Software Countermeasures for Control Flow Integrity of Smart Card C Codes"
in ESORICS'2014 (Lalande, Heydemann, Berthomé).

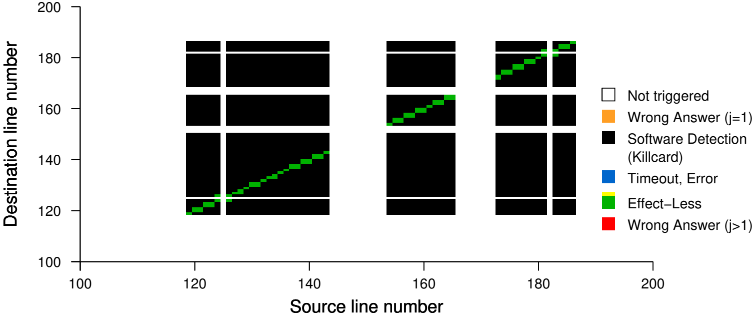
Thank you!

(Diode Laser Station from Riscure)

⇒

"Software Countermeasures for Control Flow Integrity of Smart Card C Codes"
in ESORICS'2014 (Lalande, Heydemann, Berthomé).

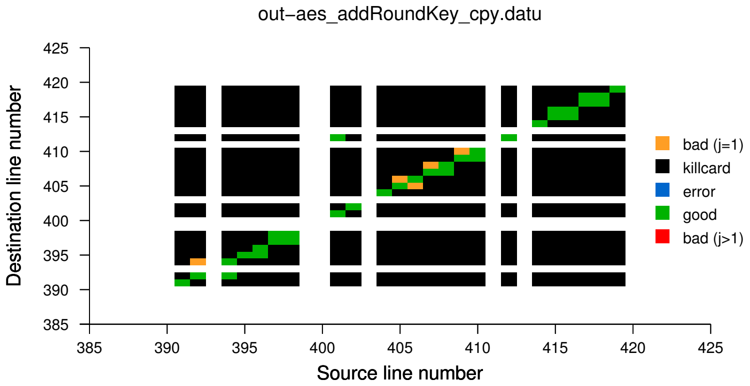
Weaknesses visualization: Secured FISSC



Visualization of verifyPIN_1 + CM (secured)

Available in FISSC !

Weaknesses visualization with CFI



Visualization of weaknesses for the secured version

Securing conditional control flow

'onditional code

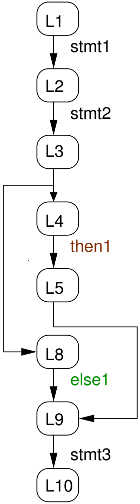
```
void f() {
    stmt1;
    smt2;:
    if (cond){
        then1;
        then2;
    }
    else
        else1;
     stmt3;
  }
```

Securing conditional control flow

'onditional code

```
   void f() {
1:      stmt1;
2       smt2;:
3:      if (cond){
4:          then1;
5;          then2;
6;      }
7:      else
8:          else1;
9:       stmt3;
10:   }
```

Securing conditional flow

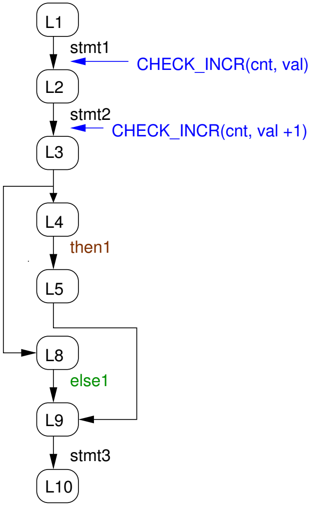


Securing conditional control flow

:onditional code

```
      void f() {
1:        stmt1;
2         smt2;:
3:        if (cond){
4:            then1;
5;            then2;
6;        }
7:        else
8:            else1;
9:         stmt3;
10:     }
```

Securing conditional flow

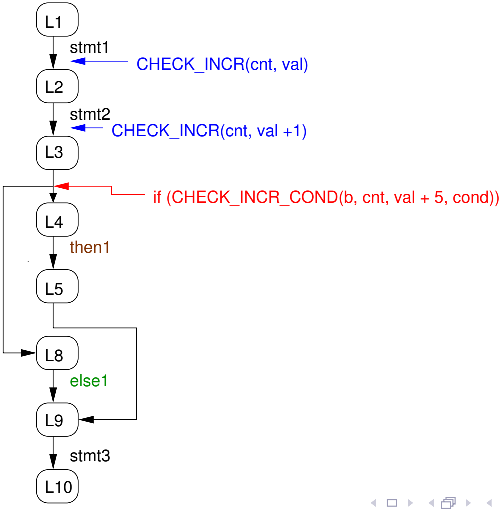


Securing conditional control flow

'onditional code

```
void f() {
    stmt1;
    smt2;:
    if (cond){
       then1;
       then2;
    }
    else
       else1;
     stmt3;
  }
```

Securing conditional flow

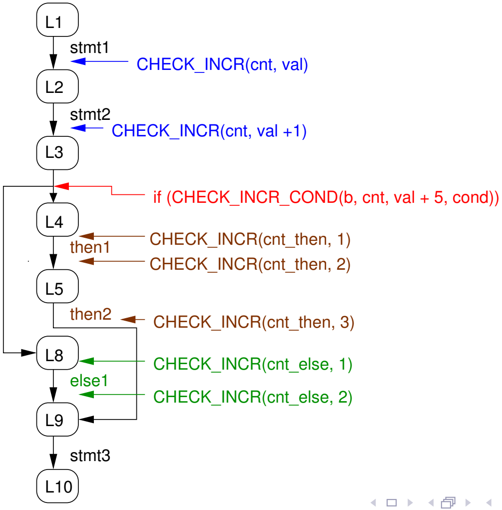


Securing conditional control flow

ʼonditional code

```
    void f() {
1:      stmt1;
2       smt2;:
3:      if (cond){
4:          then1;
5;          then2;
6;      }
7:      else
8:          else1;
9:       stmt3;
10:   }
```

Securing conditional flow

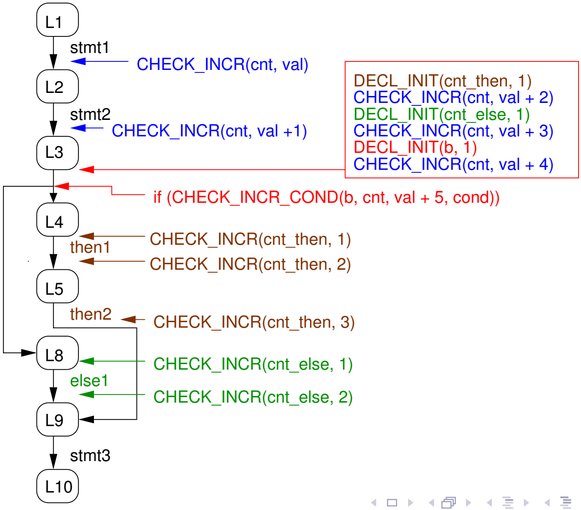


Securing conditional control flow

'onditional code

```
    void f() {
1:      stmt1;
2       smt2;:
3:      if (cond){
4:          then1;
5;          then2;
6;      }
7:      else
8:          else1;
9:       stmt3;
10:   }
```

Securing conditional flow

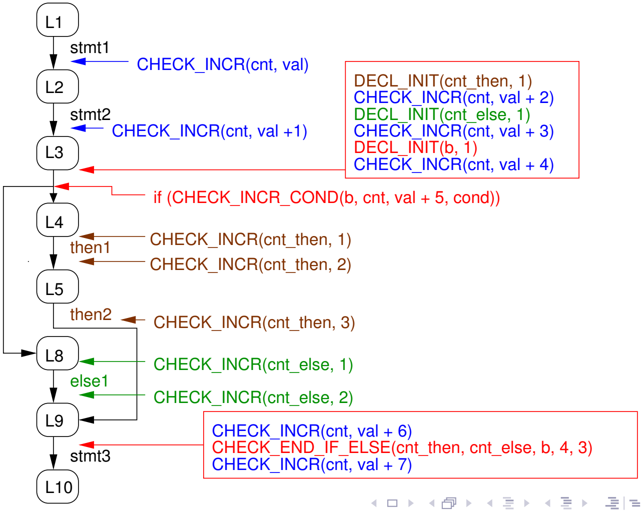


Securing conditional control flow

'onditional code

```
     void f() {
1:      stmt1;
2       smt2;:
3:      if (cond){
4:         then1;
5;         then2;
6;      }
7:      else
8:         else1;
9:       stmt3;
10:   }
```

Securing conditional flow

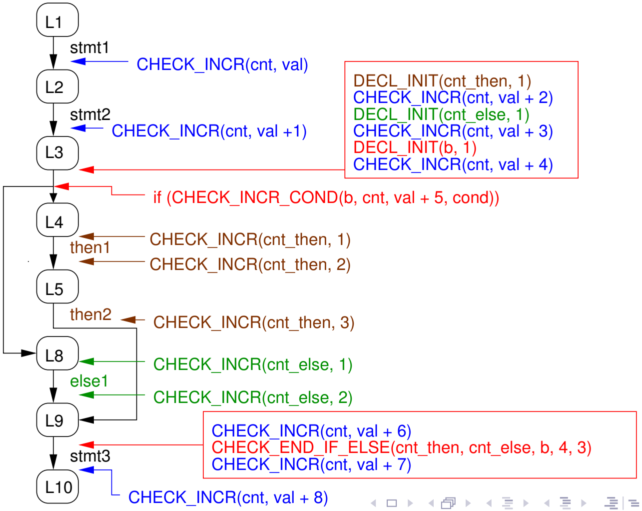


Securing conditional control flow

'onditional code

```
    void f() {
1:      stmt1;
2       smt2;:
3:      if (cond){
4:          then1;
5;          then2;
6;      }
7:      else
8:          else1;
9:       stmt3;
10:   }
```

Securing conditional flow



Security macros

Needed macro:

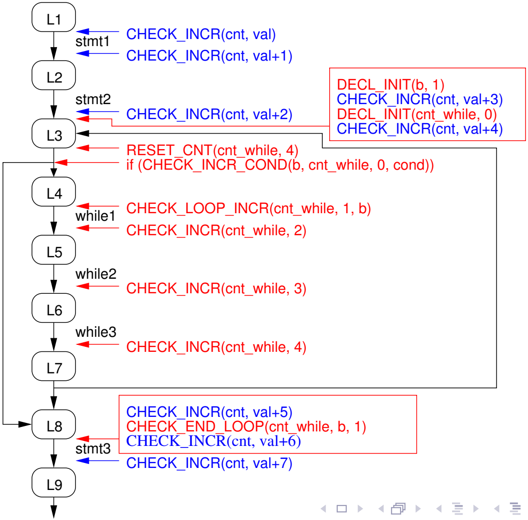
```
#define DECL_INIT(cnt, x) int cnt; if ((cnt = x) != x) killcard();

#define CHECK_INCR(cnt, x) cnt = (cnt == x ? cnt +1 : killcard());

#define CHECK_END_IF_ELSE(cnt_then, cnt_else, b, x, y) if (! ((cnt_then
      == x && cnt_else == 0 && b) || (cnt_else == y && cnt_then == 0
      && !b))) killcard();

#define CHECK_END_IF(cnt_then, b, x) if ( ! ( (cnt_then == x && b) || (
      cnt_then == 0 && !b) ) ) killcard();

#define CHECK_INCR_COND(b, cnt, val, cond) (b = (((cnt)++ != val) ?
      killcard() : cond))
```

Securing loop control flow

Loop code

```
      void f(){
        ...
L1:    stmt1;
L2:    stmt2;
L3:    while (cond){
L4:         whiile1;
L5:         while2;
L6:         while3;
L7:     }
L8:     stmt3;
L9:     ...
L10: }
```

Securing loop flow



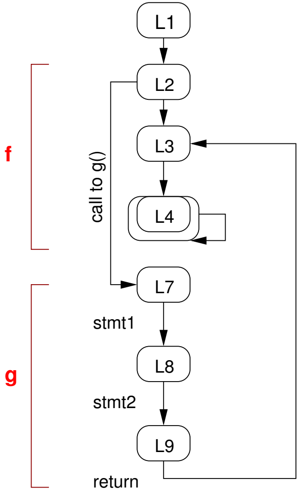
Security macros

Needed macro:

```
#define DECL_INIT(cnt, x) int cnt; if ((cnt = x) != x) killcard();

#define CHECK_INCR(cnt, x) cnt = (cnt == x ? cnt +1 : killcard());

#define CHECK_INCR_COND(b, cnt, val, cond) (b = (((cnt)++ != val) ?
    killcard() : cond))

#define CHECK_LOOP_INCR(cnt, x, b) cnt = (b && cnt == x ? cnt +1 :
    killcard());

#define CHECK_END_LOOP(cnt_while, b, val) if ( ! (cnt_while == val && !
    b) ) killcard();
```

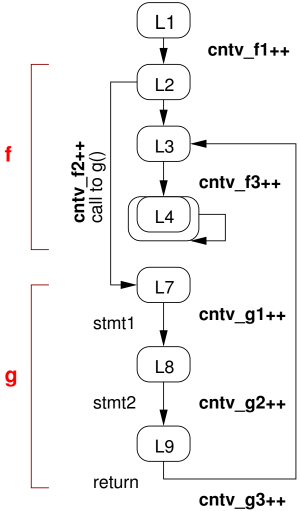
Model M: straight-line flow

```
        void f(){
L1:         ...
L2:         g();
L3:         ...
L4:     }

        void g(){
L7:         stmt1;
L8:         stmt2;
L9:         return;
        }
```

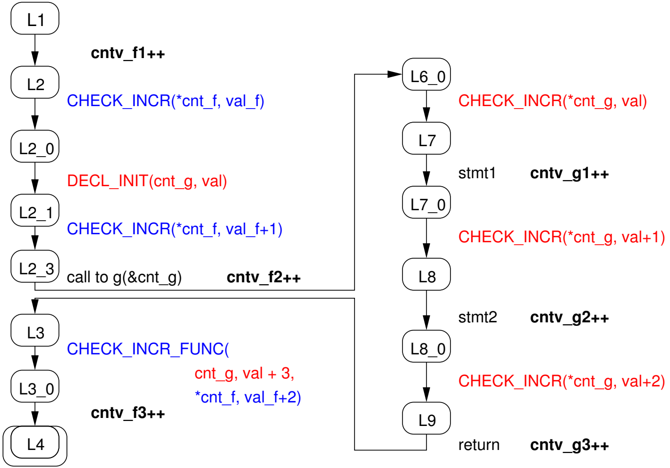


Model M: straight-line flow

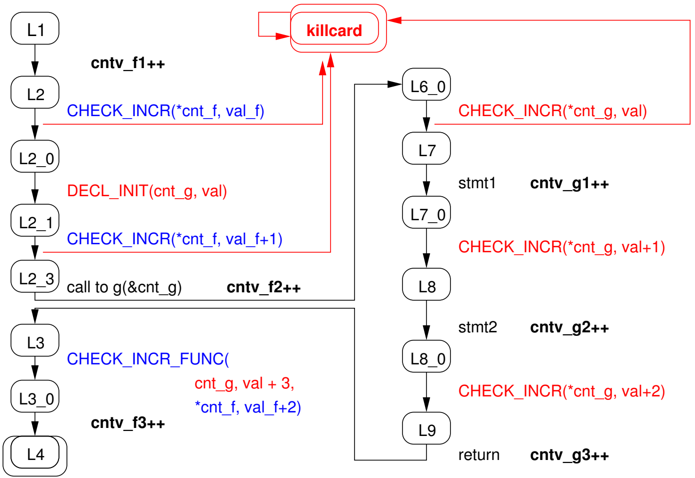
```
        void f(){
L1:        ...
L2:        g();
L3:        ...
L4:     }

        void g(){
L7:        stmt1;
L8:        stmt2;
L9:        return;
        }
```



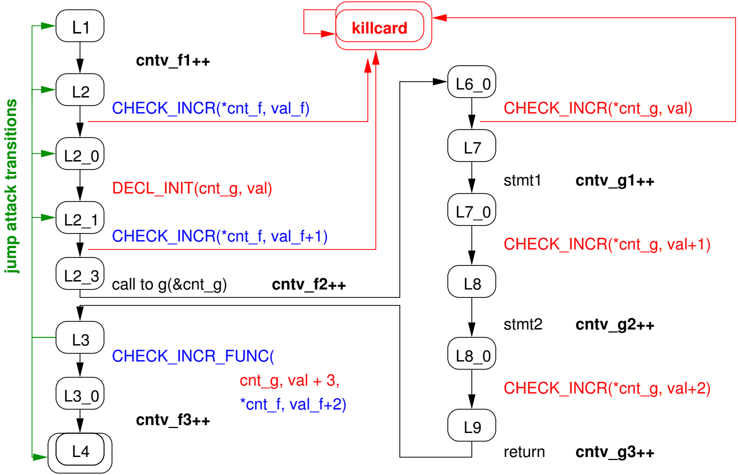
Model CM: straight-line flow



Model CM: straight-line flow

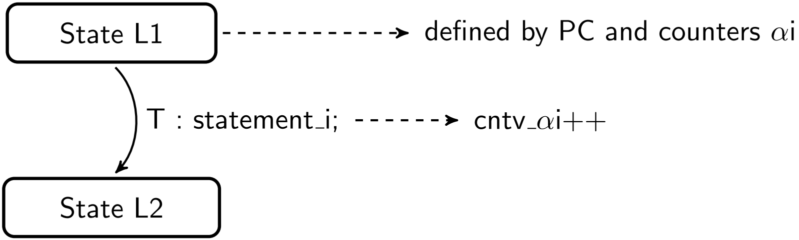


Model CM: straight-line flow



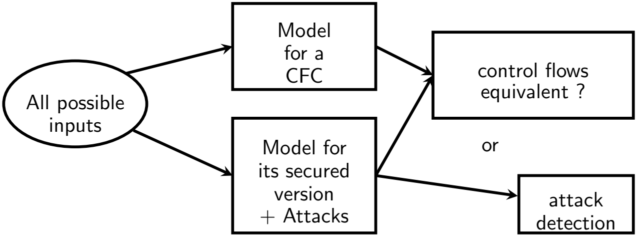
Model for one statement

In function α:



Execution of statement_i and PC is modeled by cntv_αi++

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

Properties to verify for straight-line flow case

1. Any path in `M(c)` or `CM(c)` reaches a final absorbing state.
2. The statement counter values in any final correct state in `CM(c)` (with a program counter value different from `killcard`) are equal to the statement counter values in final states of `M(c)`.
3. In `CM(c)` at any time and in any path, counters $\texttt{cntv\_}\alpha\texttt{i}$ and $\texttt{cntv\_}\alpha\texttt{(i+1)}$ for two adjacent statements `stmt_i` and `stmt_i+1` in a straight-line flow respects:

$$1 \geq \texttt{cntv\_}\alpha\texttt{(i+1)} \geq \texttt{cntv\_}\alpha\texttt{i} \geq 0$$

or execution will reach a final state with the `killcard` value for the program counter.

CTL properties to verify for straight-line flow

```
; P1 : final state reachability in M and CM
  AG(AF(M.pc=L4))
  AG(AF(CM.pc=L4 + CM.pc=killcard))

; P2 : right statement execution counts in CM and M when reaching a correct
   final state
  AG((M.pc=L4) . (CM.pc=L4) => (M.cnt_f1=CM.cnt_f1) .
     (M.cnt_f2=CM.cnt_f2) . (M.cnt_f3=CM.cnt_f3) . (M.cnt_g1=CM.cnt_g1)
     . (M.cnt_g2=CM.cnt_g2) . (M.cnt_g3=CM.cnt_g3))

; P3 : right order of statement execution in CM or attack detection
  AG(((CM.cnt_f1=CM.cnf_f2 + CM.cnt_f1=CM.cnt_f2+1) .
     (CM.cnt_f2=CM.cnf_f3 + CM.cnt_f2=CM.cnt_f3+1) .
     (CM.cnt_g1=CM_cnt_g2 + CM.cnt_g1=CM.cnt_g2+1) .
     (CM.cnt_g2=CM.cnt_g3 + CM.cnt_g2=CM.cnt_g3+1)) +
     AF(CM.pc=killcard))
```

Securing code cost - x86

Size and overhead for original and secured version (+ CM)

	x86				
	Simulation time	Size		Execution time	
		bytes	overhead	time	overhead
AES	27m	17 996		1.27 ms	
AES + CM	9h 46m	30 284	(+68%)	2.61 ms	(+106%)
SHA	1h 18m	13 235		1.47 µs	
SHA + CM	16h 52m	21 702	(+64%)	2.81 µs	(+91%)
Blowfish	5h 52m	30 103		47.6 µs	
Blowfish + CM	3d 6h 19m	46 680	(+55%)	70.6 µs	(+48%)

Securing code cost - arm-v7

Size and overhead for original and secured version (+ CM)

	arm-v7m			
	Size		Execution time	
	bytes	overhead	time	overhead
AES	4216		38.3 ms	
AES + CM	15 696	(+272%)	191.7 ms	(+400.5%)
SHA	3184		106.5µs	
SHA + CM	7752	(+143%)	499.1µs	(+368%)
Blowfish	6292		3.02 ms	
Blowfish + CM	16 396	(+161%)	6.3 ms	(+109%)