Sleuth:
Automated Verification of Software Power Analysis Countermeasures

**Ali Galip Bayrak*, Francesco Regazzoni†,
David Novo*, and Paolo Ienne***

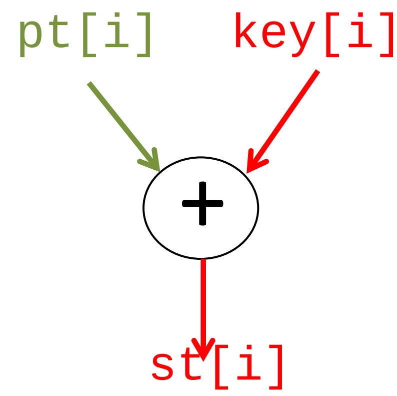
*** Ecole Polytechnique Fédérale de Lausanne (EPFL)
† TU Delft and ALaRI University of Lugano**

CHES 2013

Side-Channel Attacks and Protection

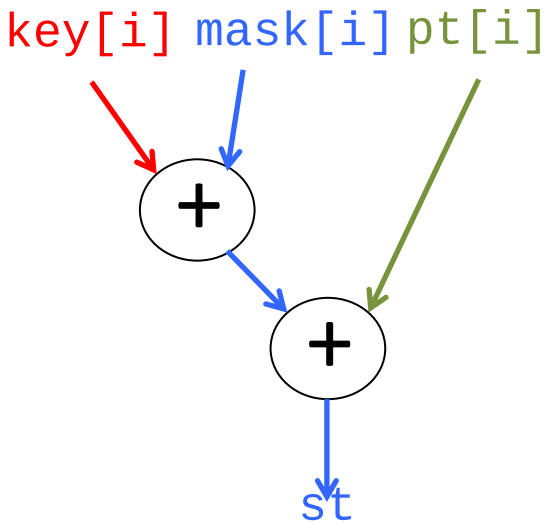
Unprotected first AddRoundKey of AES

```
void ARK() {
        unsigned char i;
        for (i=0;i<16;i++){
                st[i] = pt[i] ^ key[i];
        }
}
```



Masked first AddRoundKey of AES

```
void maskedARK() {
        unsigned char i;
        for (i=0;i<16;i++){
                st[i] = pt[i] ^
                        (key[i] ^ mask[i]);
        }
}
```

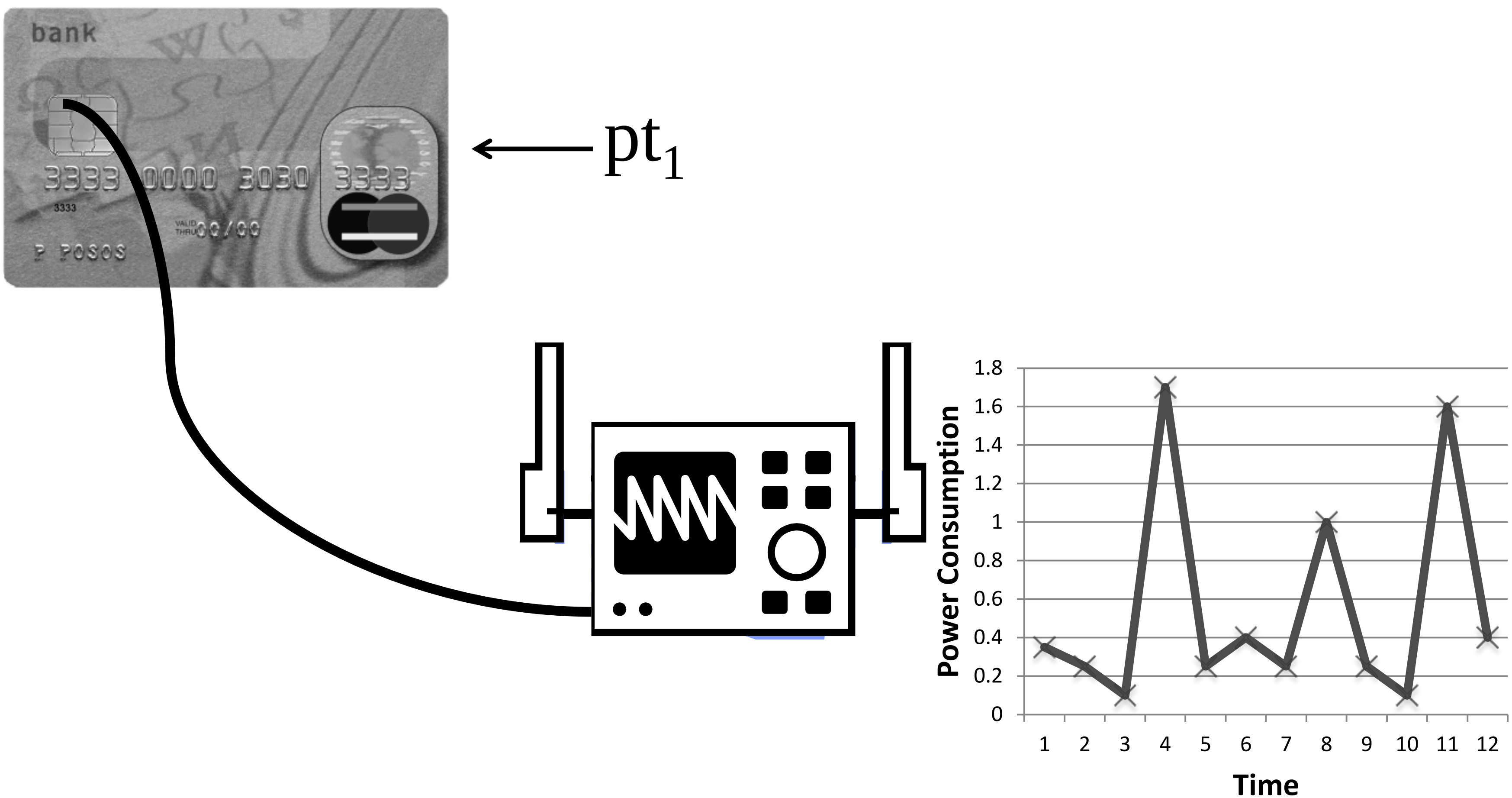


Verification (Functionality)

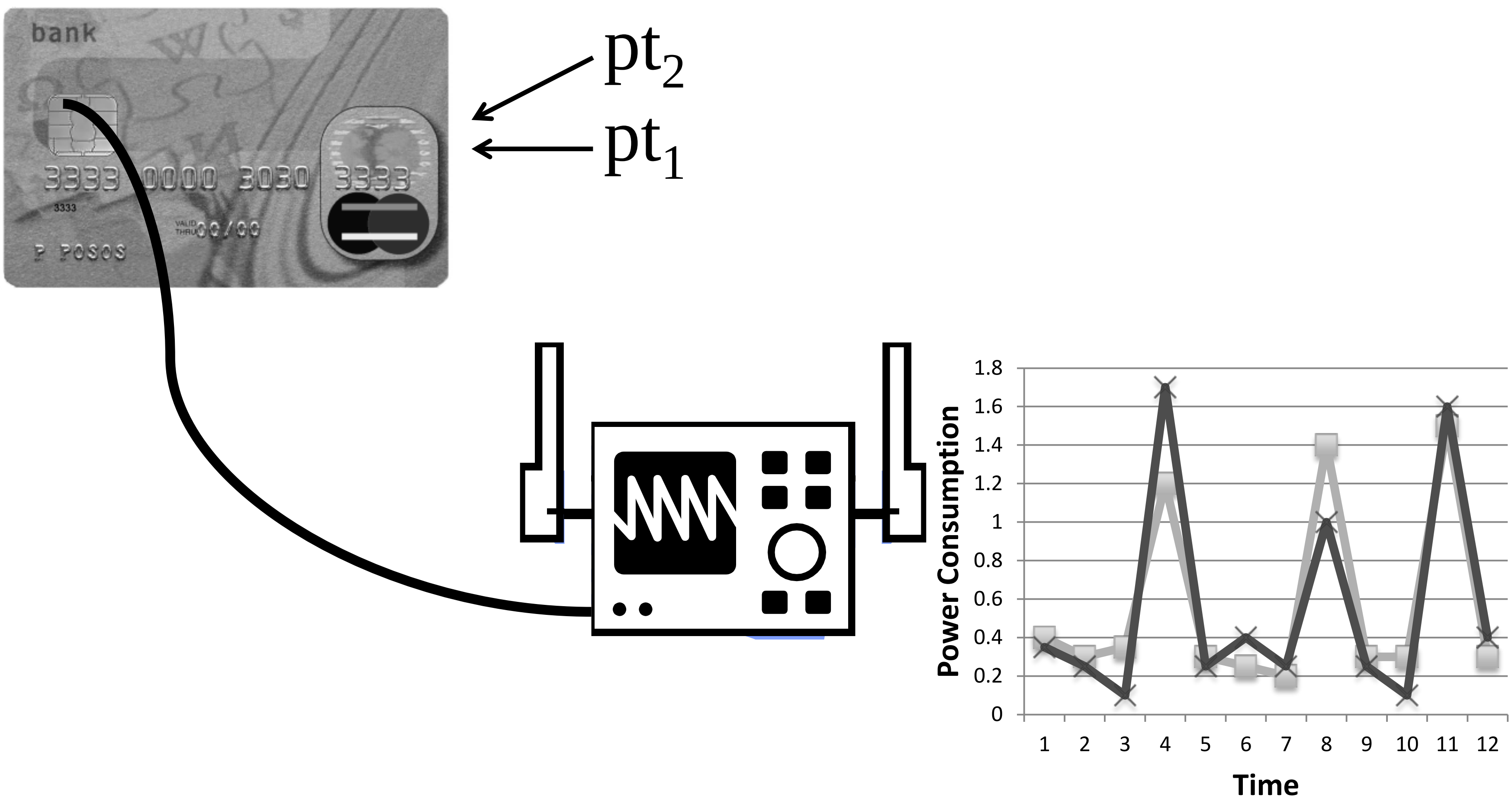

```
void maskedARK() {
    unsigned char i;
    for (i=0;i<16;i++){
        st[i] = pt[i] ^
            (key[i] ^ mask[i]);
    }
}
```

✔

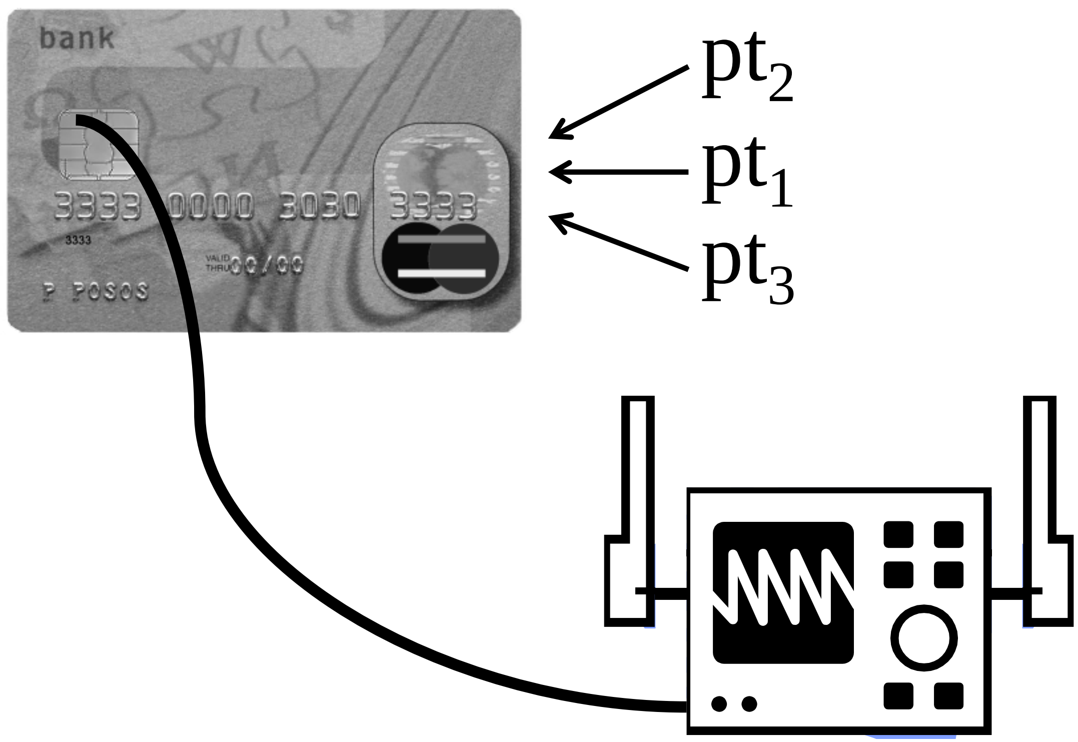
Verification (Security)

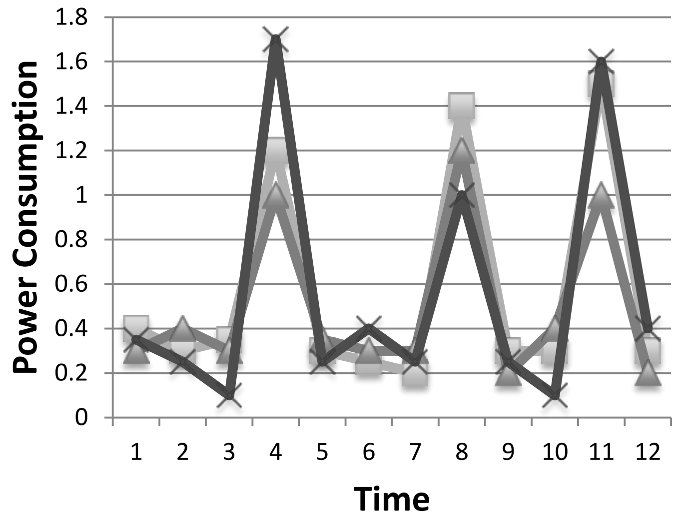


Verification (Security)

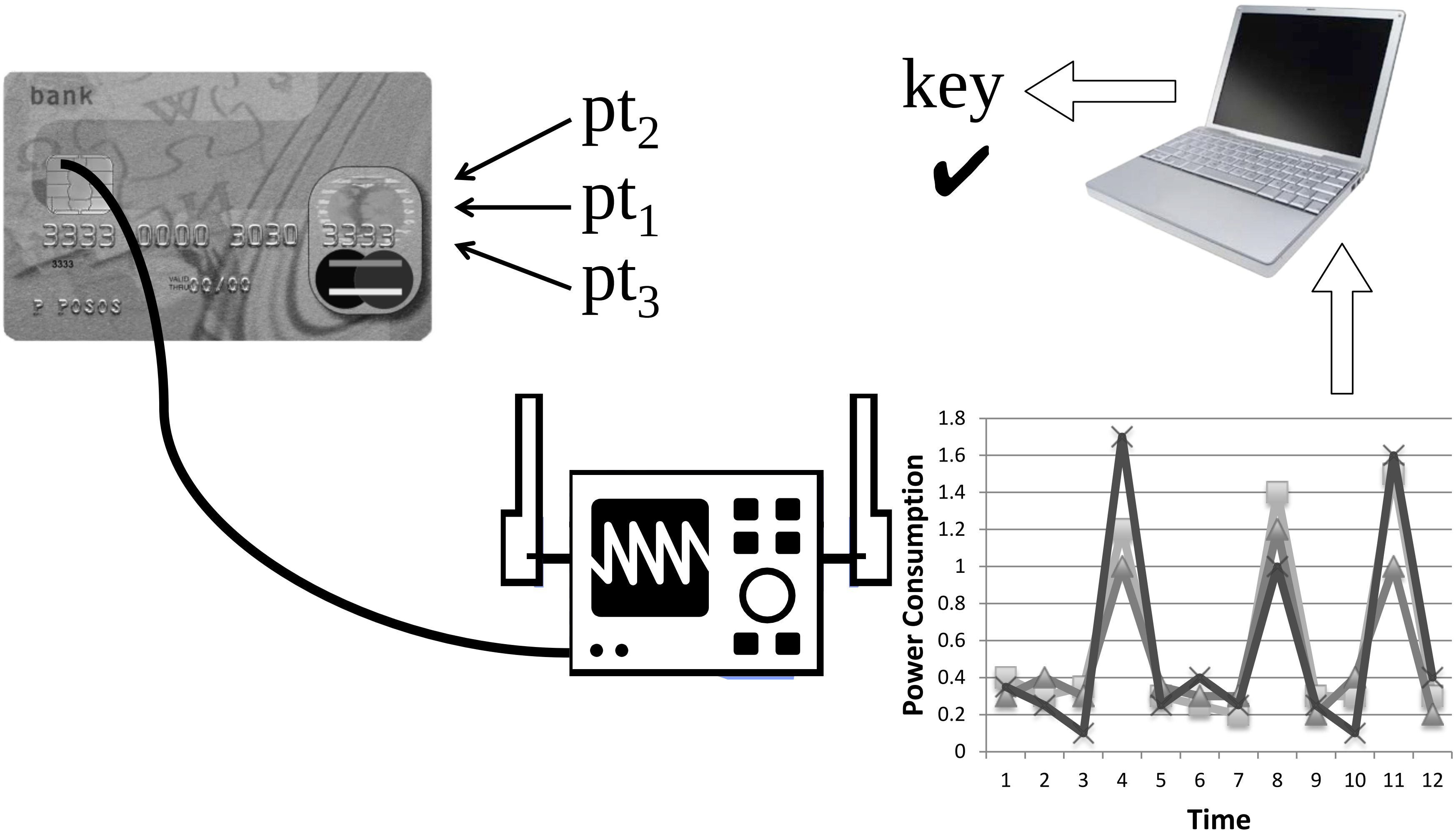


Verification (Security)





Verification (Security)



Never Trust Your Compiler

```
void maskedARK() {
  unsigned char i;
  for (i=0;i<16;i++){
    st[i] = pt[i] ^
      (key[i] ^ mask[i]);
  }
}
```

avr-gcc-4.5.3 -O3

```
       .text
.global ARK
       .type   ARK, @function
ARK:
/* prologue: function */
/* frame size = 0 */
/* stack size = 0 */
.L__stack_usage = 0
       lds r24,key
       lds r25,pt
       eor r24,r25
       lds r25,mask
       eor r24,r25
       sts st,r24
       lds r24,key+1
       lds r25,pt+1
       eor r24,r25
       ...
```

Verification is Important

```
void maskedARK() {
  unsigned char i;
  for (i=0;i<16;i++){
    st[i] = pt[i] ^
     (key[i] ^ mask[i]);
  }
}
```

→ **Sleuth** →

```
lds r24,key
lds r25,pt
eor r24,r25
```

Find the sensitive operations of a given program.

Outline

- Sensitivity definitions
- Methodology
- Experimental studies

Sensitivity Definitions

- **Goal:** Given a program, find the sensitive operations, which leak critical information.
- **Definitions we need:**
  - Program
  - Types (secret, public, random)
  - Leakage
  - Sensitivity

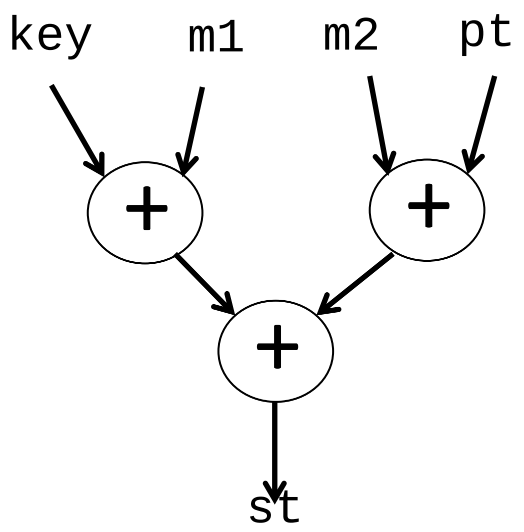
Program

- A sequence of
  - branch-free
  - three-address form
  - arithmetic/logic or memory operations.

- Example:

```
key_r = key   xor m1
pt_r  = pt    xor m2
st    = pt_r  xor key_r
```

Inputs: `key, pt, m1, m2`



Program

- A sequence of
  - branch-free
    - static analysis is exponentially complex for input-dependent branches.
    - many countermeasures (e.g., masking, random precharging) do not use such branches.
  - three-address form
    - *x* = *y op z*, *x* = *y[z]*, or *y[z]* = *x*.
    - for simplicity of representation.
  - arithmetic/logic or memory operations.

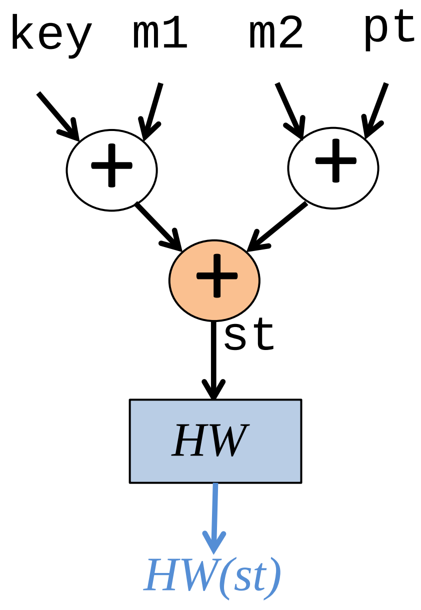
Type

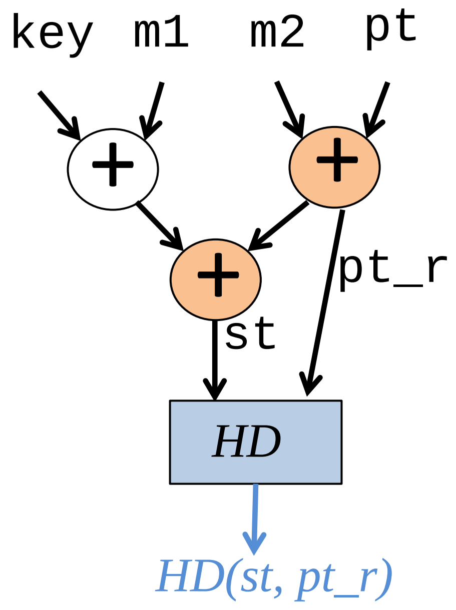
- Each input is tagged with one of the types
  - ***secret*:** content should not be revealed (e.g., key).
  - ***public*:** content is observable by third-party (e.g., plaintext).
  - ***random*:** uniformly distributed random values (e.g., mask).
- Example:

```
key : secret
pt  : public
m1  : random
m2  : random
```

Leakage Model

- A model of the side-channel leakage (e.g., power consumption) of a device *h*, for a given subset of operations *d* of a program *p*.

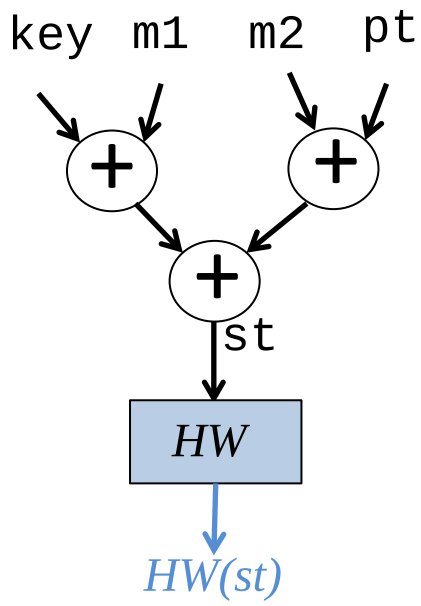




Sensitivity

- For a given
  - program *p* whose input variables $\{v_0,\ldots,v_{k-1}\}$ have types $\{t_0,\ldots,t_{k-1}\}$,
  - a device *h*,
  - a leakage model *l*,
  - ***sensitivity*** of a subset *d'* of operations of *p* represents whether leakage *l(d', p, h)* depends on at least one *secret* variable but not on any *random* variable.

Sensitivity



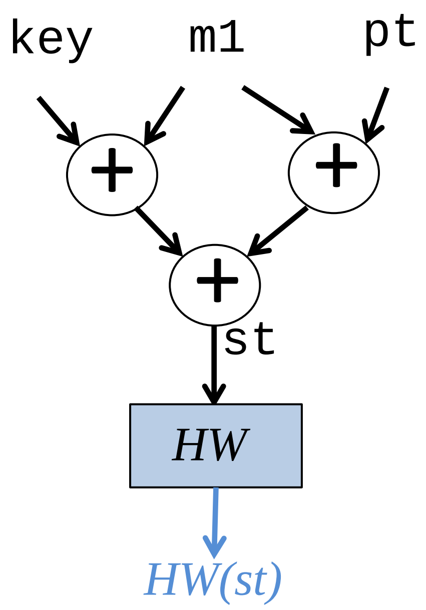
$HW(st) = HW(key \oplus m1 \oplus pt \oplus m2)$

$HW(st) \sim key$? : yes

$HW(st) \sim m1$? : yes

not sensitive

Sensitivity



$HW(st) = HW(key \oplus pt)$

$HW(st) \sim key\ ?$: yes

$HW(st) \sim m1\ ?$: no

sensitive

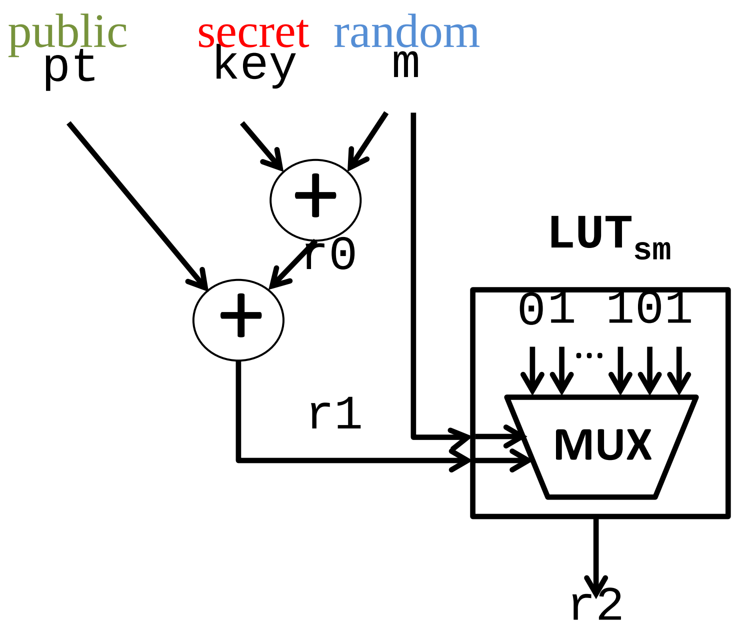
Methodology

- Represent the program as a graph.

- Use satisfiability queries to detect the dependencies and sensitivity.

Graph Representation

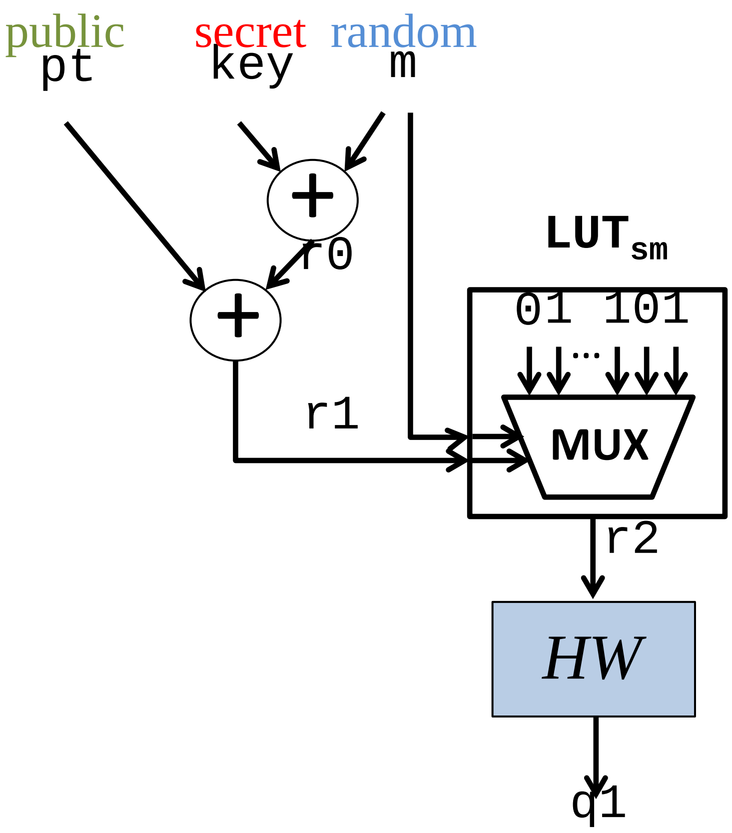
```
r0 = key ^ m;
r1 = pt ^ r0;
r2 = sm[r1];
```



Sample implementation in C

Graph representation

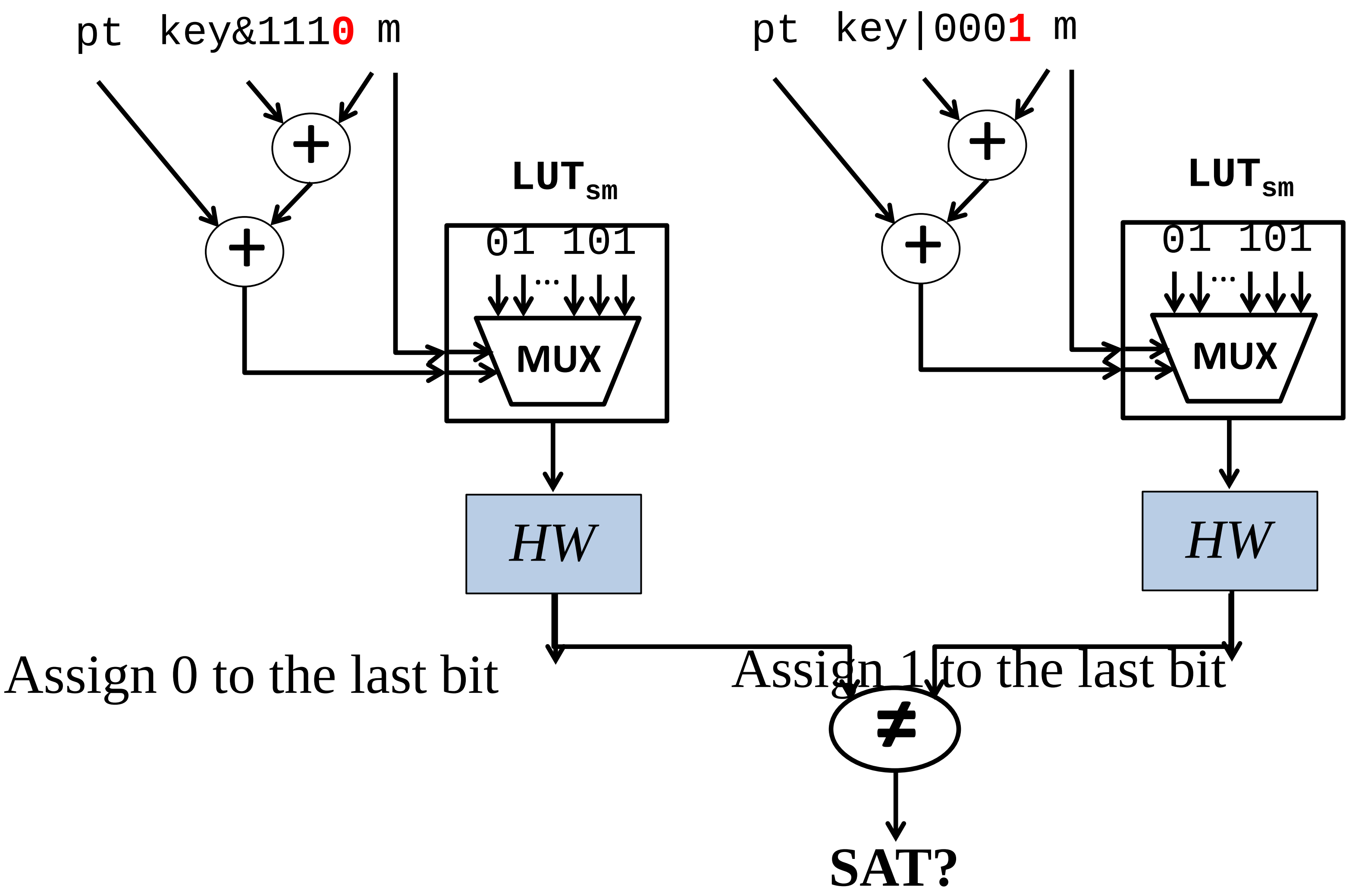
Sensitivity Detection



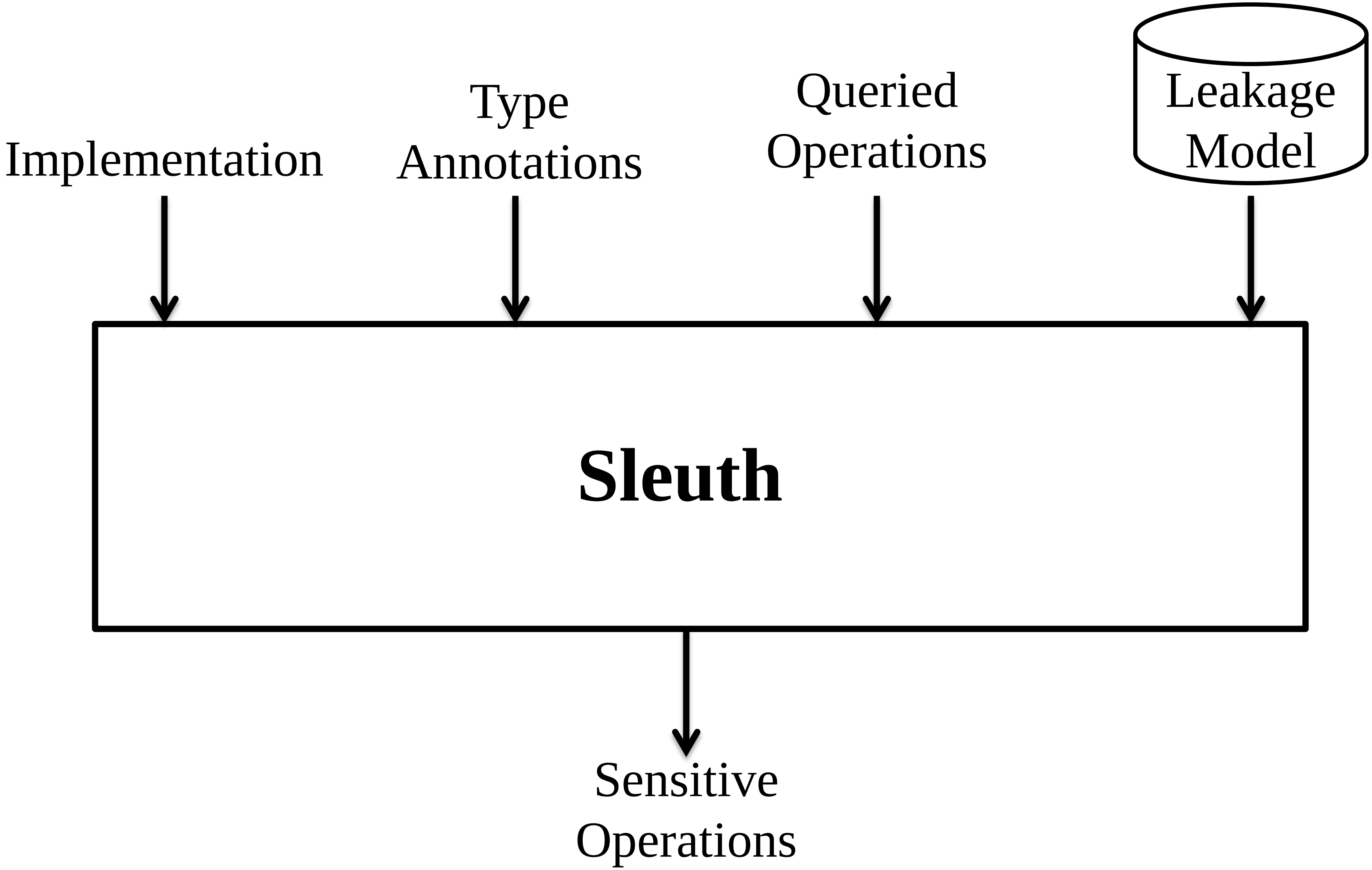
Is q_1 sensitive?

- $q_1 \sim$ key?
- $\neg (q_1 \sim m)$?

Dependency Check



Sleuth


Implementation
Type
Annotations
Queried
Operations
Leakage
Model
Sleuth
Sensitive
Operations

Experimental Studies

- Compilers are not perfect.
- Programmers are not perfect.
- Countermeasures are not perfect.

Compiler Related Problems

```
unsigned char st[16];
unsigned char key[16];
unsigned char pt[16];
unsigned char mask[16];
void ARK() {
  unsigned char i;
  for (i=0 ; i<16 ; i++) {
    st[i] = pt[i] ^
      (key[i] ^ mask[i]);
  }
}
```

```
  .text
.global ARK
  .type
ARK, @function
ARK:
/□ prologue: function □/
/□ frame size = 0 □/
/□ stack size = 0 □/
.L__stack_usage = 0
  lds r24,key
  lds r25,pt
  eor r24,r25
  lds r25,mask
  eor r24,r25
  sts st,r24
  lds r24,key+1
  lds r25,pt+1
  eor r24,r25
  ...
```

- We used *Hamming weight* univariate leakage model.
- Detects all 16 such problems in 0.02 seconds.
- Similar problems arise in later operations (e.g., MixColumns).

  It takes 430 seconds to detect all problems in a round of AES.

Programmer Related Problems

```
// swap st[2] with st[10]
tmp = st[2];
st[2] = st[10];
st[10] = tmp;
```

```
1 lds r24,st+2
2 sts tmp,r24
3 lds r25,st+10
4 sts st+2,r25
5 sts st+10,r24
```

- In the Boolean masking algorithm of Herbst et al. [19], st[2] and st[10] use the same mask.
- We used *Hamming distance* leakage model.
- Bivariate leakage will be

$$HD(st[2], st[10]) = HD(st\_orig[2] \oplus m, st\_orig[10] \oplus m)$$

$$= HW(st\_orig[2] \oplus st\_orig[10]).$$

- Finds all such problems (i.e., also between line 4 and 5) in 477 seconds.

Countermeasure Related Problems

Find *A* such that $x' \square r_x = A + r_x$

```
BooleanToArithmetic_Messerges(x',r_x) {
   // randomly select: C = 0 or C = -1
   B = C □ r_x; /* B=r_x or B=r̅_x */
   A = B □ x'; /* A=x or A=x̅ */
   A = A - B;  /* A=x - r_x or A=x̅ - r̅_x */
   A = A + C;  /* A=x - r_x or A=x̅ - r̅_x */
   A = A □ C;  /* A=x - r_x */
}
```

- If we use a proper leakage model an attack is possible [Coron and Goubin ‘00].
- We used *parity of the result* as a leakage model.
- Finds in 0.02 seconds.

Conclusions and Discussions

- Our SAT-based methodology is generic (does not depend on the algorithm or countermeasure).
- We can find crucial real world problems (that can invalidate the countermeasure) in a reasonable time.
- The user can make use of an extendible library of leakage models.