SmartCard Lab: Final Presentation

Group 2

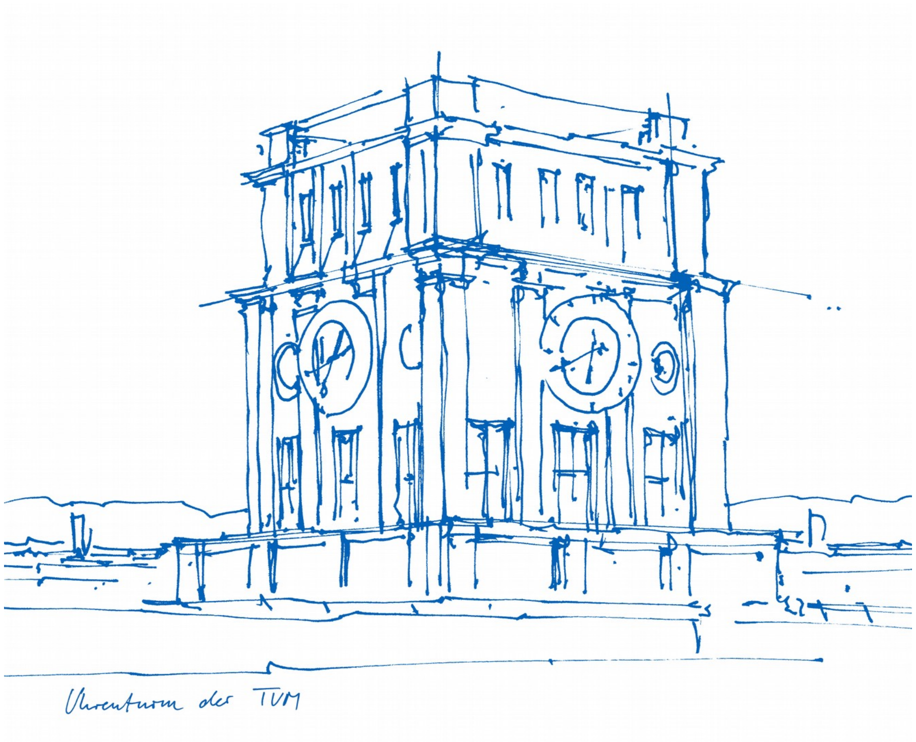
Martin Achtner

Ridon Arifi

Ines Ben Hmida

Maximilian Galanis

Simon Ulshöfer



Outline

- Team B – Countermeasures
  - Implemented Countermeasures
  - Impact of Countermeasures
  - Random Number Generation
- Team A – DPA
  - Improvements
  - Countermeasures
  - Benchmarks
- Project Plan

Team B – Countermeasures

- Implemented Countermeasures
- Impact of Countermeasures
- Random Number Generation

Implemented Countermeasures

- Masking
- Hiding
  - Shuffling
  - NOPs

Masking

Precomputation:

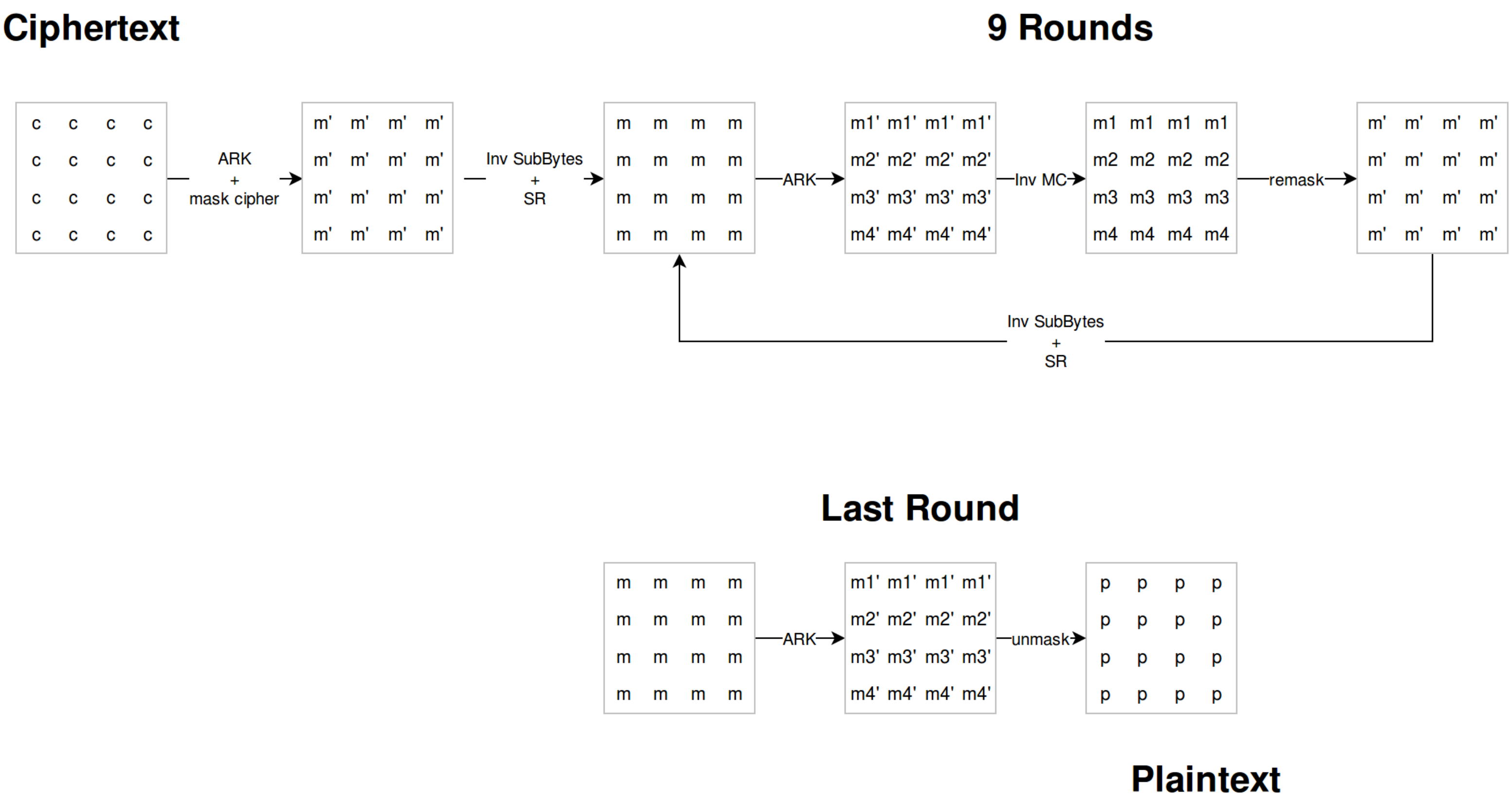
- Generate „input and output masks“ m, m’
- Precompute masked inv. S-Box S’(x): S‘(x ⊕ m‘) = S(x) ⊕ m
- Generate „transformed masks“ m1‘, m2‘, m3‘, m4‘ for inv. MixColumns
- Precompute „input masks“ using inv. MC : (m1, m2, m3, m4) = inv. MC(m1’, m2’, m3’, m4’)

Masking

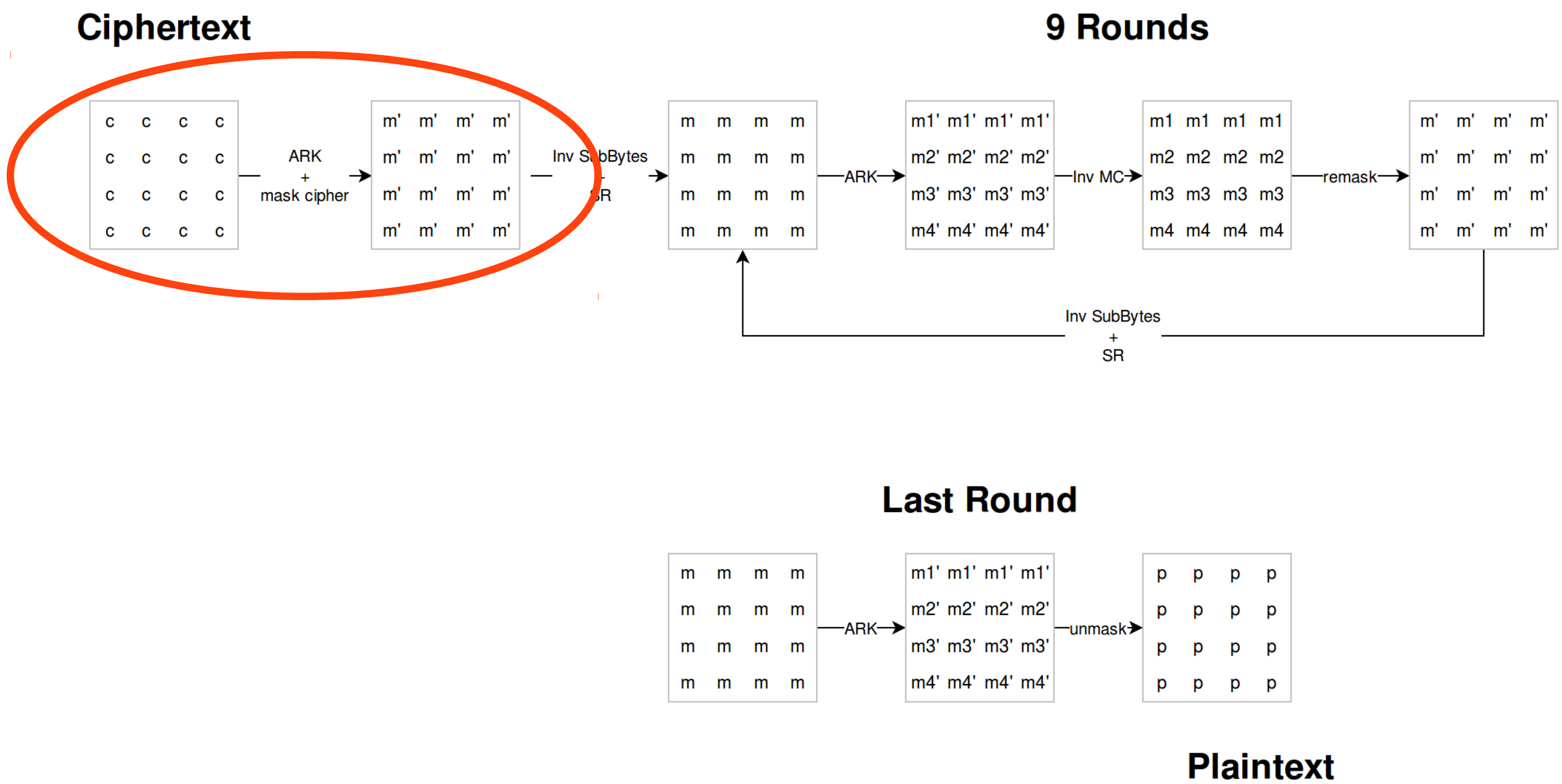
```
void mask_init(void){
        // Generate input and output masks
        //init_rand();

        m_i  = get_rand();
        mp_i = get_rand();
        m1_t = get_rand();
        m2_t = get_rand();
        m3_t = get_rand();
        m4_t = get_rand();

        // MixColumn
        m1_i = mult_14[m1_t] ^ mult_11[m2_t] ^ mult_13[m3_t] ^ mult_9[m4_t];
        m2_i = mult_9[m1_t]  ^ mult_14[m2_t] ^ mult_11[m3_t] ^ mult_13[m4_t];
        m3_i = mult_13[m1_t] ^ mult_9[m2_t]  ^ mult_14[m3_t] ^ mult_11[m4_t];
        m4_i = mult_11[m1_t] ^ mult_13[m2_t] ^ mult_9[m3_t]  ^ mult_14[m4_t];

        // compute masked sbox Sica. 5b.page 12
        for(uint16_t i = 0; i < 256; i++){
                inv_sbox_masked[(i ^ mp_i)] = (inv_sbox[i] ^ m_i);
        }
}
```

m_i → m
mp_i → m'
m1_t → m1'
...

Masking


Ciphertext
9 Rounds
c c c c
ARK + mask cipher
m' m' m' m'
Inv SubBytes + SR
m m m m
ARK
m1' m1' m1' m1'
m2' m2' m2' m2'
m3' m3' m3' m3'
m4' m4' m4' m4'
Inv MC
m1 m1 m1 m1
m2 m2 m2 m2
m3 m3 m3 m3
m4 m4 m4 m4
remask
m' m' m' m'
Inv SubBytes + SR
Last Round
m m m m
ARK
m1' m1' m1' m1'
m2' m2' m2' m2'
m3' m3' m3' m3'
m4' m4' m4' m4'
unmask
p p p p
Plaintext

Masking

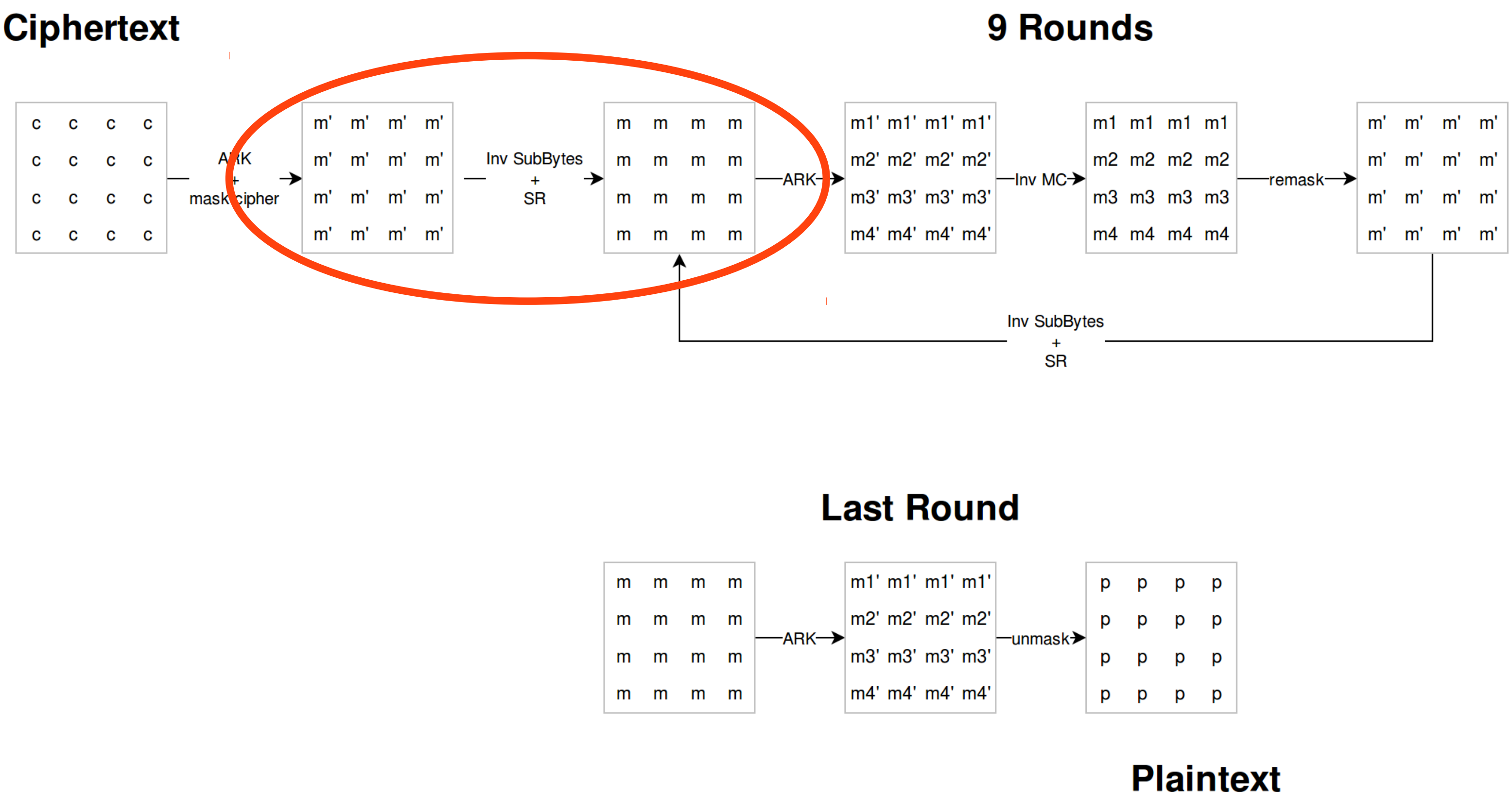

Ciphertext
9 Rounds
ARK
+
mask cipher
Inv SubBytes
+
SR
ARK
Inv MC
remask
Inv SubBytes
+
SR
Last Round
ARK
unmask
Plaintext

Masking

```
void mask_key(uint8_t *key_buffer){
	for(uint8_t i = 0; i < AES128_NUM_OF_BYTES; i++){
			if(i%4 == 0)
				key_buffer[i] ^= m1_t ^ m_i;
			else if(i%4 == 1)
				key_buffer[i] ^= m2_t ^ m_i;
			else if(i%4 == 2)
				key_buffer[i] ^= m3_t ^ m_i;
			else
				key_buffer[i] ^= m4_t ^ m_i;
		}
}
```

```
void mask_ciphertext(uint8_t *cipher){
	for(uint8_t i = 0; i < AES128_NUM_OF_BYTES; i++){
			if(i%4 == 0)
				cipher[i] ^= m1_t ^ mp_i ^ m_i;
			else if(i%4 == 1)
				cipher[i] ^= m2_t ^ mp_i ^ m_i;
			else if(i%4 == 2)
				cipher[i] ^= m3_t ^ mp_i ^ m_i;
			else
				cipher[i] ^= m4_t ^ mp_i ^ m_i;

	}
}
```

Masking


Ciphertext
9 Rounds
ARK + mask cipher
Inv SubBytes + SR
ARK
Inv MC
remask
Inv SubBytes + SR
Last Round
ARK
unmask
Plaintext
c c c c
m' m' m' m'
m m m m
m1' m1' m1' m1'
m2' m2' m2' m2'
m3' m3' m3' m3'
m4' m4' m4' m4'
m1 m1 m1 m1
m2 m2 m2 m2
m3 m3 m3 m3
m4 m4 m4 m4
p p p p

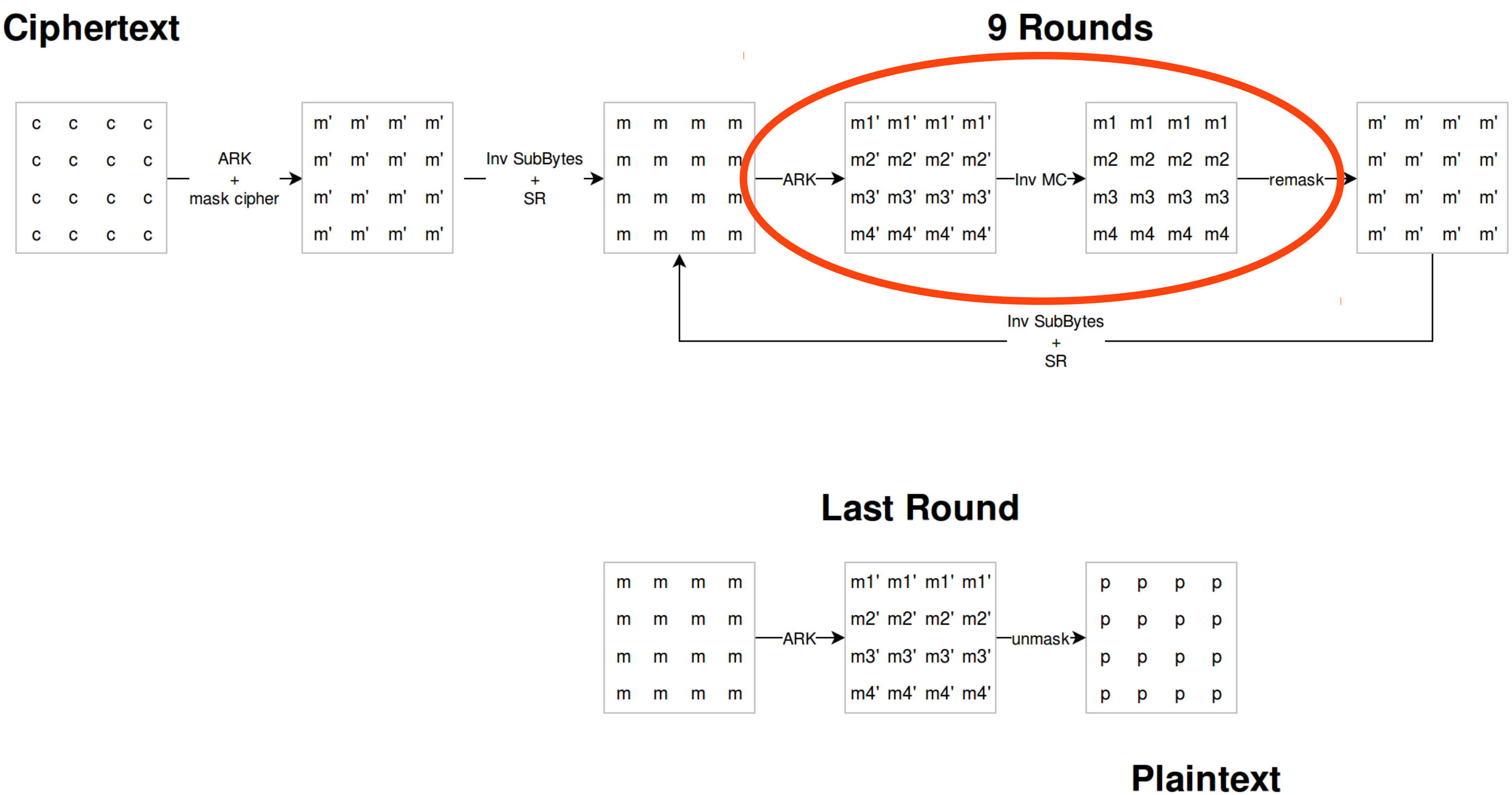
Masking

```
void mask_init(void){
        // Generate input and output masks
        //init_rand();

        m_i  = get_rand();
        mp_i = get_rand();
        m1_t = get_rand();
        m2_t = get_rand();
        m3_t = get_rand();
        m4_t = get_rand();

        // MixColumn
        m1_i = mult_14[m1_t] ^ mult_11[m2_t] ^ mult_13[m3_t] ^ mult_9[m4_t];
        m2_i = mult_9[m1_t]  ^ mult_14[m2_t] ^ mult_11[m3_t] ^ mult_13[m4_t];
        m3_i = mult_13[m1_t] ^ mult_9[m2_t]  ^ mult_14[m3_t] ^ mult_11[m4_t];
        m4_i = mult_11[m1_t] ^ mult_13[m2_t] ^ mult_9[m3_t]  ^ mult_14[m4_t];

        // compute masked sbox Sica. 5b.page 12
        for(uint16_t i = 0; i < 256; i++){
                inv_sbox_masked[(i ^ mp_i)] = (inv_sbox[i] ^ m_i);
        }
}
```

m_i → m
mp_i → m’
m1_t → m1’
…

Lehrstuhl für Sicherheit in der Informationstechnik
Fakultät für Elektrotechnik
Technische Universität München

Masking

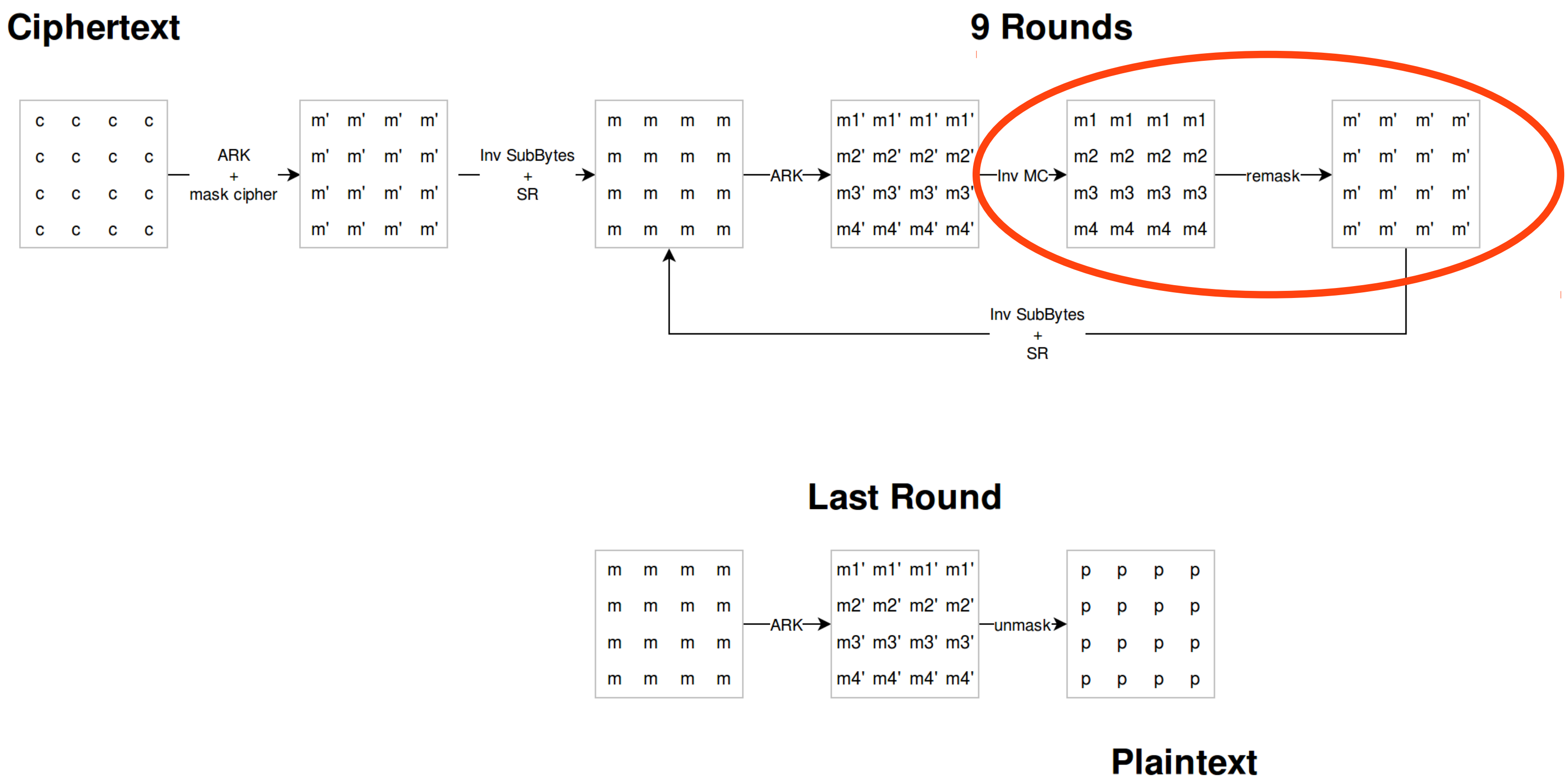


Masking

```
void mask_init(void){
        // Generate input and output masks
        //init_rand();

        m_i  = get_rand();
        mp_i = get_rand();
        m1_t = get_rand();
        m2_t = get_rand();
        m3_t = get_rand();
        m4_t = get_rand();

        // MixColumn
        m1_i = mult_14[m1_t] ^ mult_11[m2_t] ^ mult_13[m3_t] ^ mult_9[m4_t];
        m2_i = mult_9[m1_t]  ^ mult_14[m2_t] ^ mult_11[m3_t] ^ mult_13[m4_t];
        m3_i = mult_13[m1_t] ^ mult_9[m2_t]  ^ mult_14[m3_t] ^ mult_11[m4_t];
        m4_i = mult_11[m1_t] ^ mult_13[m2_t] ^ mult_9[m3_t]  ^ mult_14[m4_t];

        // compute masked sbox Sica. 5b.page 12
        for(uint16_t i = 0; i < 256; i++){
                inv_sbox_masked[(i ^ mp_i)] = (inv_sbox[i] ^ m_i);
        }
}
```

(m1, …, m4) = MC(m1’, …, m4’)

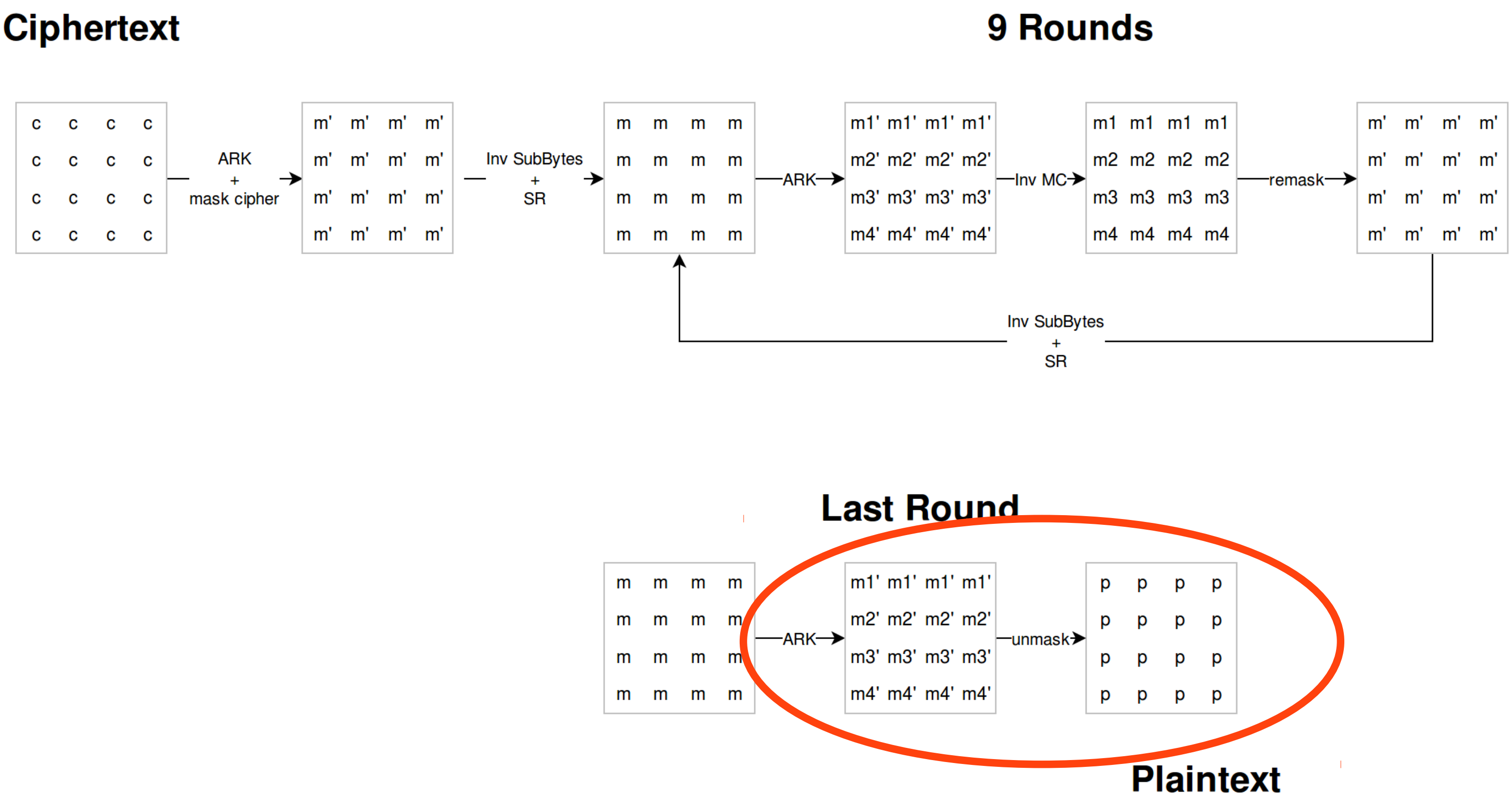
Masking


Ciphertext
9 Rounds
c c c c
ARK + mask cipher
m' m' m' m'
Inv SubBytes + SR
m m m m
ARK
m1' m1' m1' m1'
m2' m2' m2' m2'
m3' m3' m3' m3'
m4' m4' m4' m4'
Inv MC
m1 m1 m1 m1
m2 m2 m2 m2
m3 m3 m3 m3
m4 m4 m4 m4
remask
m' m' m' m'
Inv SubBytes + SR
Last Round
m m m m
ARK
m1' m1' m1' m1'
m2' m2' m2' m2'
m3' m3' m3' m3'
m4' m4' m4' m4'
unmask
p p p p
Plaintext

Masking

```
void remask(uint8_t *state){
        for(uint8_t i = 0; i < AES128_NUM_OF_BYTES; i++){
                if(i%4 == 0)
                        state[i] ^= m1_i ^ mp_i;
                else if(i%4 == 1)
                        state[i] ^= m2_i ^ mp_i;
                else if(i%4 == 2)
                        state[i] ^= m3_i ^ mp_i;
                else
                        state[i] ^= m4_i ^ mp_i;
        }
}
```

Masking


Ciphertext
9 Rounds
ARK + mask cipher
Inv SubBytes + SR
ARK
Inv MC
remask
Inv SubBytes + SR
Last Round
ARK
unmask
Plaintext

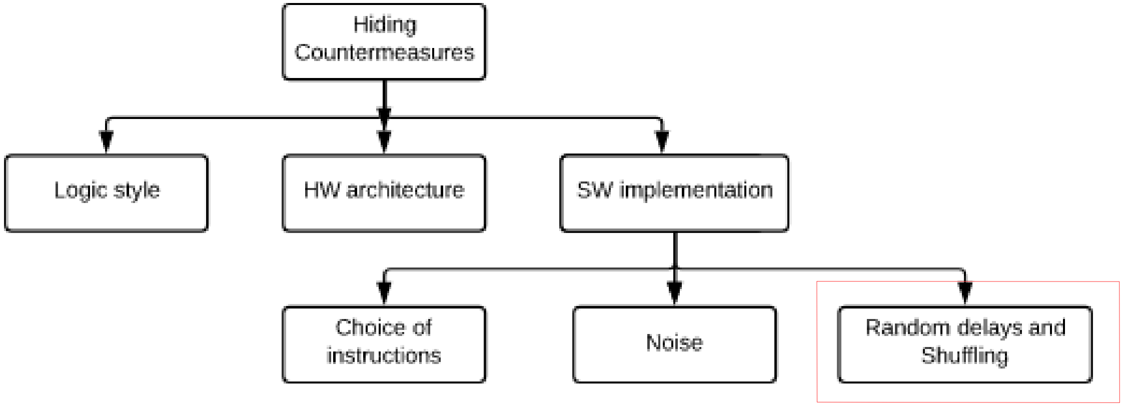
Masking

```
void unmask_plaintext(uint8_t * state){
        for(uint8_t i = 0; i < AES128_NUM_OF_BYTES; i++){
                if(i%4 == 0)
                        state[i] ^= m1_t;
                else if(i%4 == 1)
                        state[i] ^= m2_t;
                else if(i%4 == 2)
                        state[i] ^= m3_t;
                else
                        state[i] ^= m4_t;
        }
}
```

Implemented Countermeasures

Hiding power consumption

- Principle: Altering the power consumption of the Smartcard



Implemented Countermeasures

NOPs

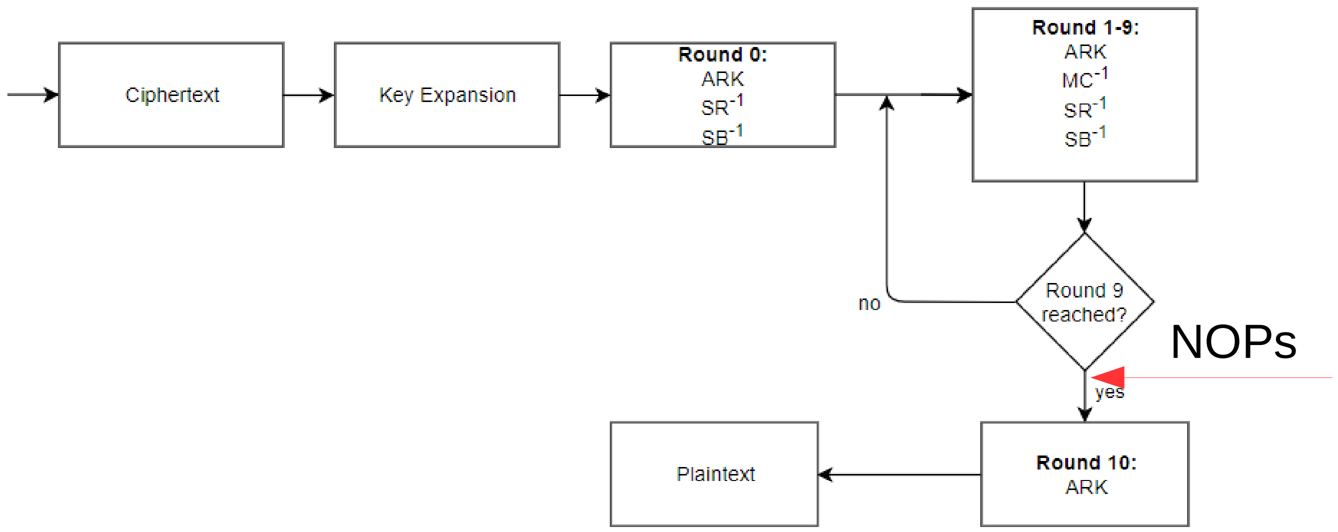
- Principle: Inserting a random number of No operations → Random delays
- Implementation: assembly no operations instruction implemented before last round of AES

```c
void no_operations(void)
{
    int random_number;
    int i;
    random_number =get_rand() & 0x0f;

    for(i=0; i<random_number; i++)
    {

        asm volatile ("nop");
    }
}
```

```c
#ifdef NOP
    no_operations();
#endif

    // Last round
    // AddRoundKey
    int_param = int_param - 16;
    ark(int_buffer, int_param);
    #ifdef MASK
    unmask_plaintext(buffer);
    #endif
}

/**
```

Implemented Countermeasures

NOPs



Lehrstuhl für Sicherheit in der Informationstechnik
Fakultät für Elektrotechnik
Technische Universität München

Implemented Countermeasures

<u>Shuffling</u>

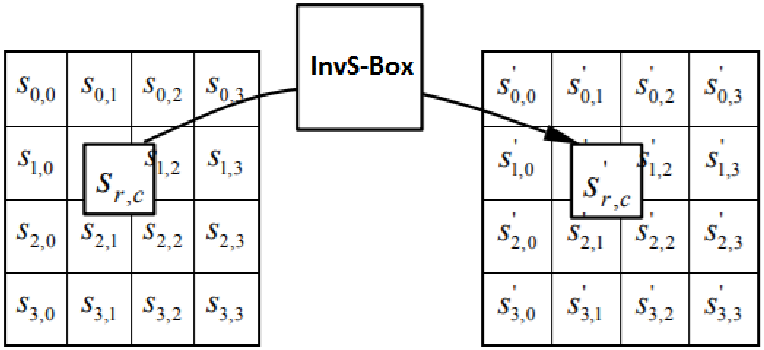
- Principle: Randomly change the sequence of operations performed in arbitrary order.
- Implementation: **Fisher-Yates Shuffle** algorithm within subbytes table lookups.

```c
void shuff_operations(uint8_t *shuff_tab){

    int i;
    int intermediate;
    int rand_index;
    for (i = 16; i > 0; i--)
    {

        rand_index= (get_rand() %i);

        intermediate = shuff_tab[i-1];
        shuff_tab[i-1] = shuff_tab[rand_index];
        shuff_tab[rand_index] = intermediate;
    }
}
```

```c
 - state: current state
 */
void subbytes(uint8_t * state){
#ifdef SHUFFLE
    uint8_t rand_array[16] = {0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15};
    shuff_operations(rand_array);
#endif

    for(uint8_t j = 0; j < AES128_NUM_OF_BYTES; j++){
#ifdef NOP
        no_operations();
#endif

#ifdef SHUFFLE
    #ifdef MASK
        state[rand_array[j]] = inv_sbox_masked[state[rand_array[j]]];
    #else
        state[rand_array[j]] = inv_sbox[state[rand_array[j]]];
    #endif
#else
    #ifdef MASK
        state[j] = inv_sbox_masked[state[j]];
    #else
        state[j] = inv_sbox[state[j]];
    #endif
#endif
    }
```

Implemented Countermeasures

Shuffling implemented in the subbytes function



Ordered Lookups from 0 to 15

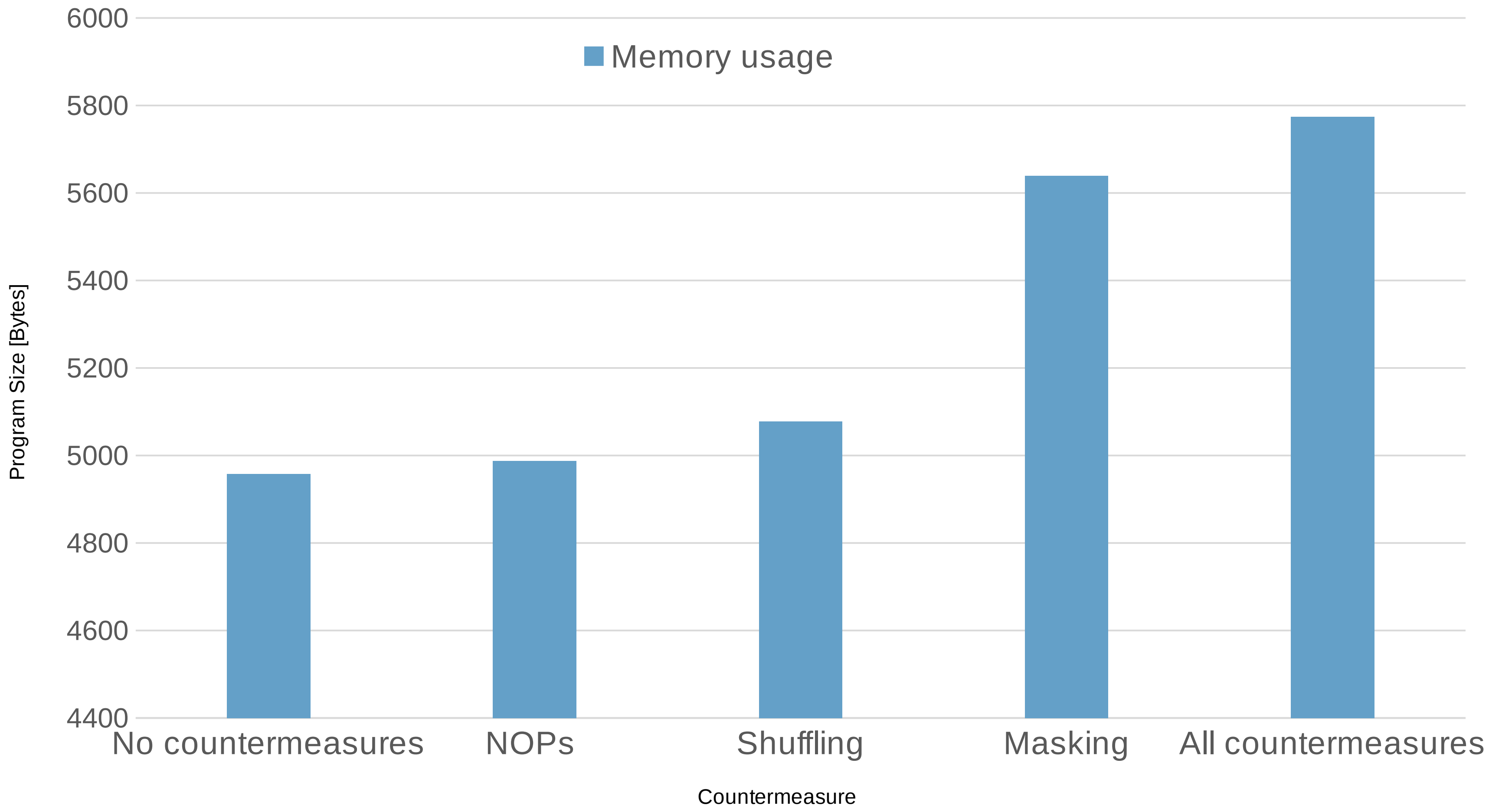
Randomly Mixed Order Lookups
→ new level of randomization
for every subbytes round

Implemented Countermeasures

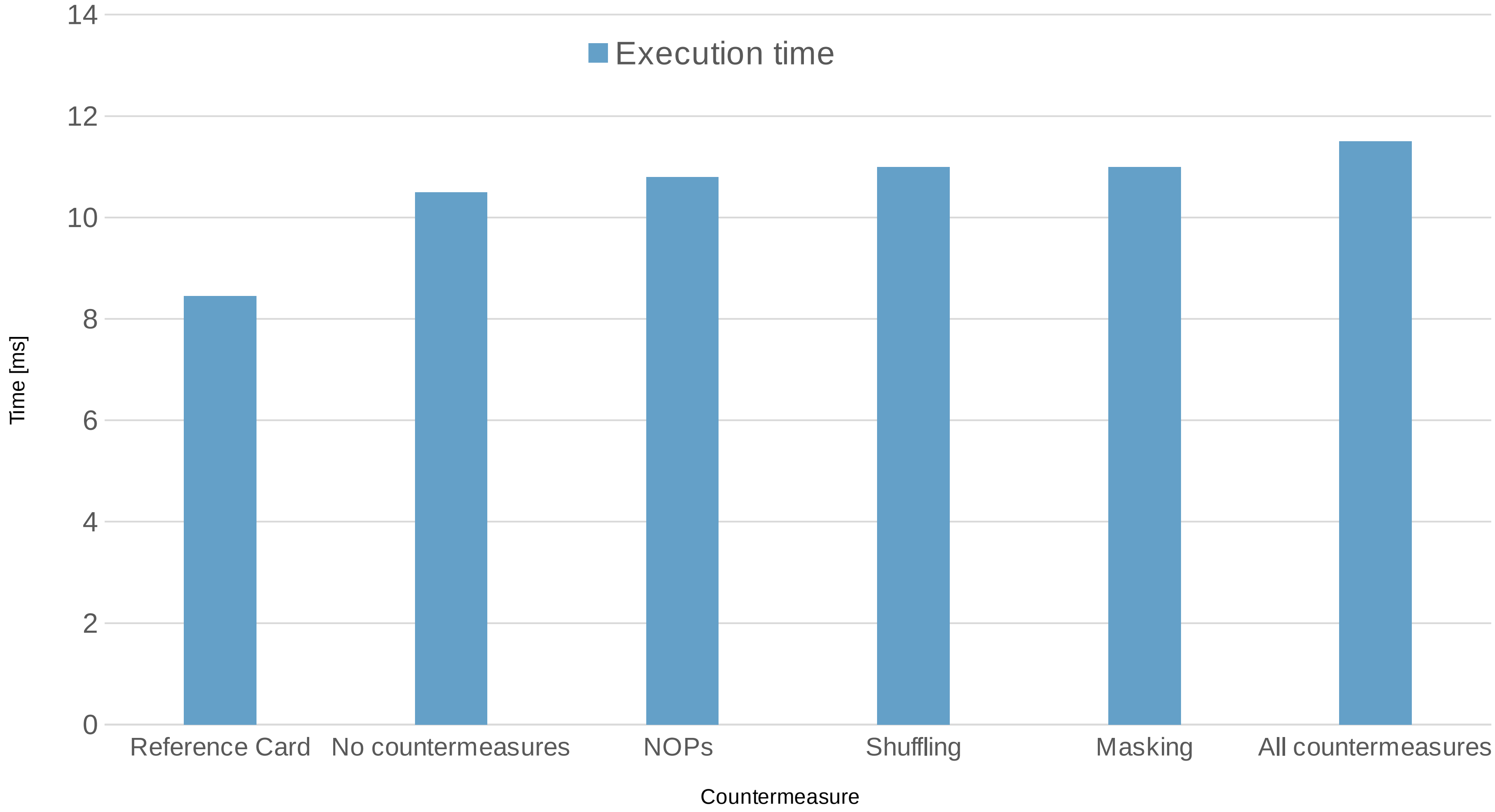
About the effectiveness of Hiding Countermeasures

Advantages	Disadvantages
• Generic • Throughput unchanged with Shuffling • Correlation linearly reduced → quadratic increase of the number of needed measurements for the attacker	• If windowing by the attacker → linear increase of effort • Alignment of power traces could remove the countermeasures

Impact of countermeasures



Impact of countermeasures



Random Number Generation

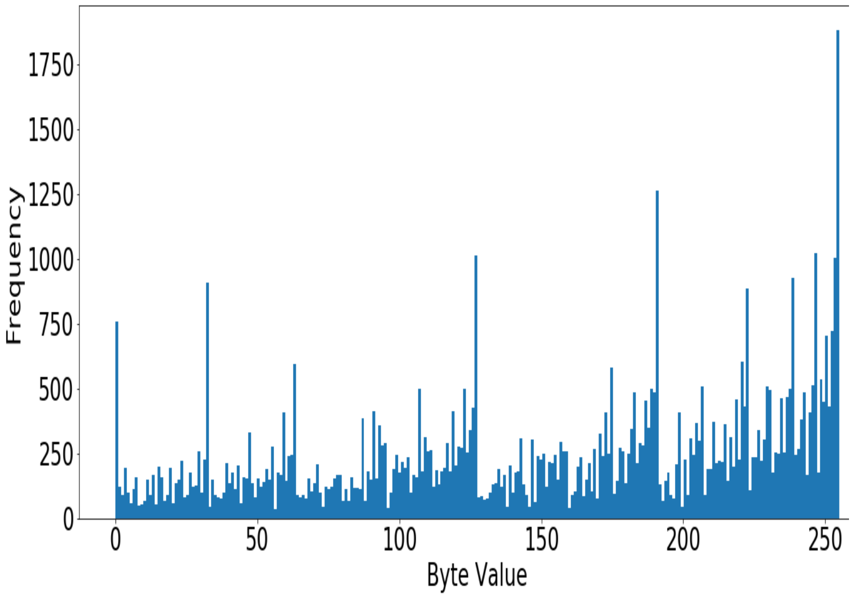
- True RNG
  - Implementation
  - Statistical Tests

- Pseudo RNG
  - Implementation
  - Statistical Tests

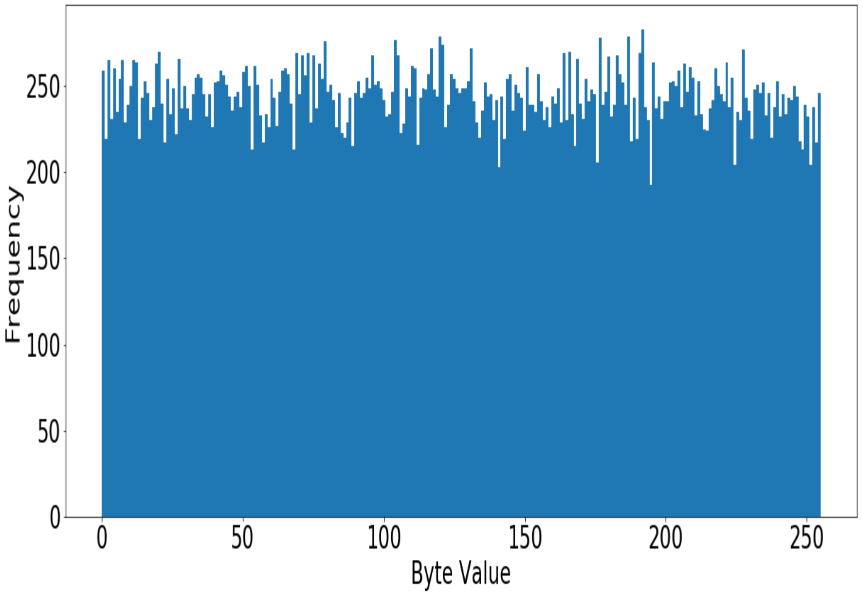
TRNG – Source

- LSB of ADC is influenced by noise
- Use multiple LSBs to generate a random byte
- Use one or more of those random bytes as seed for PRNG

TRNG – Distribution



Using LSB directly



With Von-Neumann Correction

TRNG – Tests

- Generated 500.000 Bits and send them to PC via UART
- Applied NIST Test Suite with 1, 10 and 20 streams on the file
- All tests passed
- Entropy: 7.9969 bits per byte (Tested with program “ent”)

PRNG – Source Code

```
uint8_t get_rand()
{
x++;
a = (a^c^x);
b = (b+a);
c = (c+(b>>1)^a);
return(c);
}
```

- Three seeds required
- Fast and easy algorithm
- Used for Shuffling, NOPs and masks

PRNG – Statistics

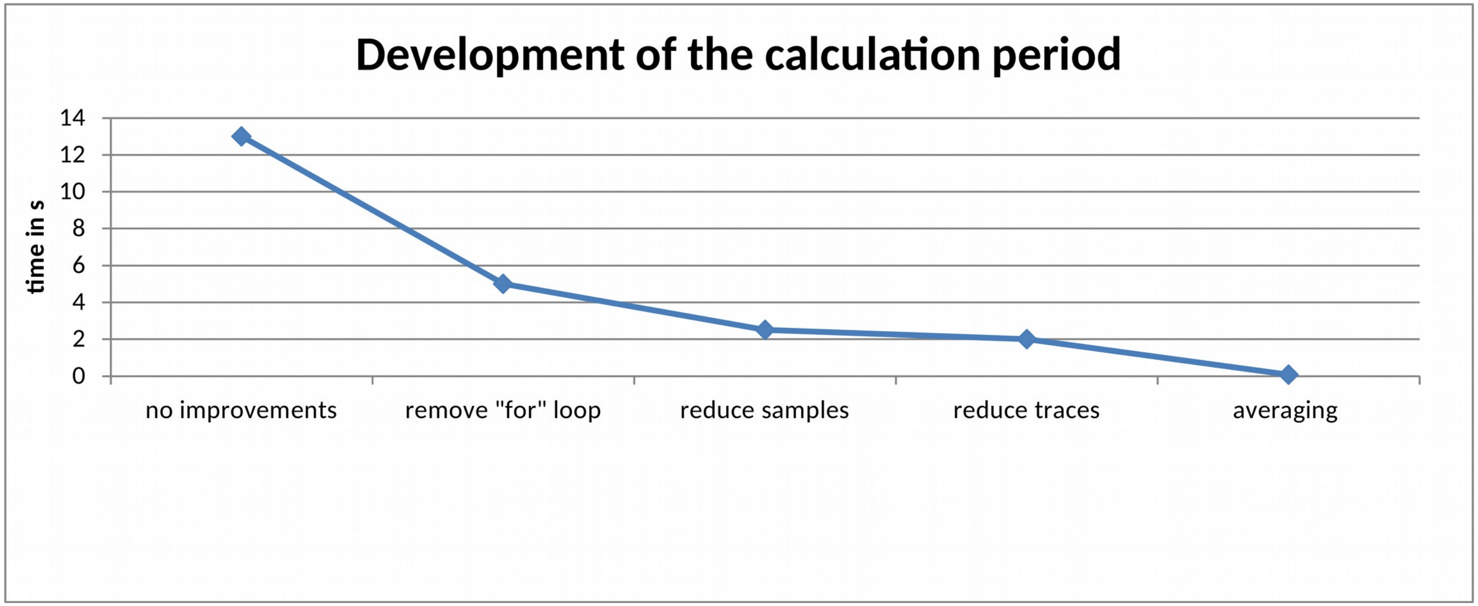
- Created 1 million bits
- Tested with 1, 10 and 20 streams
- All tests passed, results of similiar quality as reference RNG (Blum-Blum-Shub)
- Entropy: 7.9985 bits per byte (Tested with program “ent”)

Team A – DPA

- Improvements
- Countermeasures
- Benchmarks

Speed Improvements

- Massive speed improvements by getting rid of for loops
    - Replace them by using vector operations
- Trace compression



Higher-Order DPA

- **Problem**: Power consumption does not depend on unmasked intermediate values anymore
- **Solution:** Combine values in the power trace → $|HW(u_m) - HW(v_m)|$
  - Idea:
    $$u_m \oplus v_m = (u \oplus m) \oplus (v \oplus m) = u \oplus v$$
  - New hypothesis:
    $$H = HW(u \oplus v)$$
- Practical problems:
  - quadratic effort in preprocessing
  - Huge amount of traces and samples

Memory Management

- **Problems:**
  - File alone barely fits into memory
  - Even more memory needed during correlation

- **Solution:**
  1. Load small amount of samples from the file into memory (~1000)
  2. Correlate on the segment → Save key hypothesis with highest correlation for every key byte
  3. Repeat 2. until all samples have been processed
  4. Extract key hypothesis for every byte with highest correlation from all segments

- **Performance:**
  - 50000 samples in ~ 23 min on ULV CPU
  - Memory usage of ~ 500 MB

Preprocessing

- First: calculation new number of samples $n_{HODPA} = \frac{(n-1)*n}{2}$

X1	X2	X3	X4

|X1-X2|	|X1-X3|	|X1-X4|	|X2-X3|	|X2-X4|	|X3-X4|

- Check results by applying Second-Order DPA on unprotected smart card
  -> decreases correlation, but still works

Trace alignment

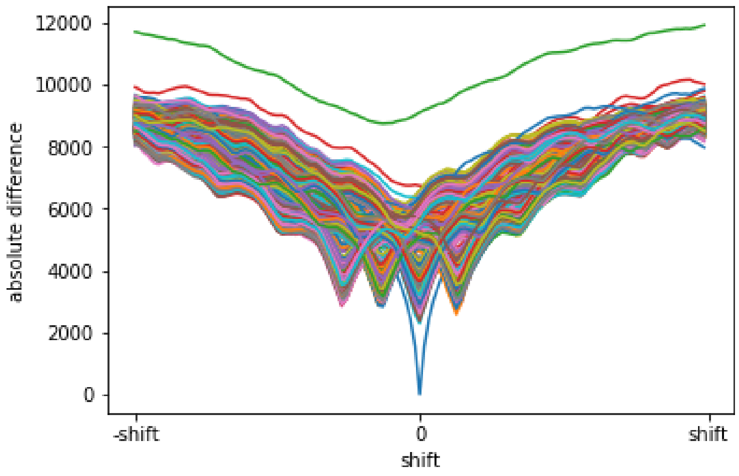
- X-Values = comparison trace (first trace)
- Y-Values = trace
- Choose the shift in percent (should be about 100 samples)

Y1	Y2	Y3	Y4	Y5	Y6	Y7	Y8	Y9	Y10				
	Y1	Y2	Y3	Y4	Y5	Y6	Y7	Y8	Y9	Y10			
		X1	X2	X3	X4	X5	X6	X7	X8	X9	X10		
			Y1	Y2	Y3	Y4	Y5	Y6	Y7	Y8	Y9	Y10	
				Y1	Y2	Y3	Y4	Y5	Y6	Y7	Y8	Y9	Y10

- Find lowest absolute difference of all shifts
- Less samples after trace alignment

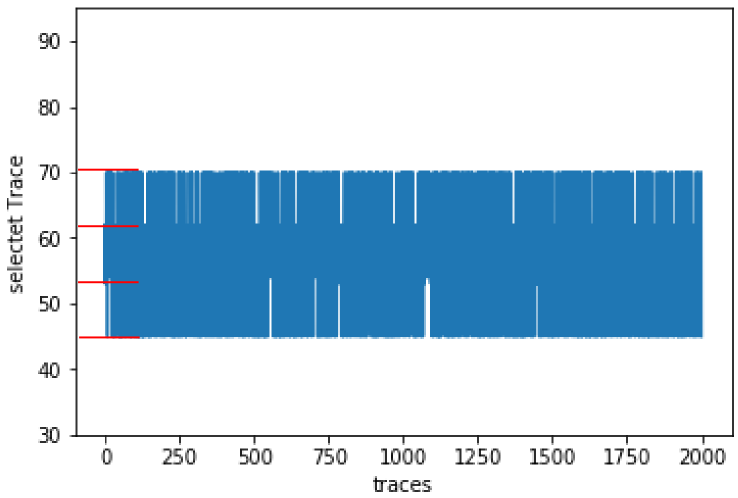
Trace alignment

Best correlation 3 nops



- Absolute difference for 3 nops
- t(total) = **80,306s**

Selected trace 3 nops



- Mostly discreet jumps

Benchmarks: no decryption

- Best correlation (~ 0,5):
  - Traces = 400 (given), Samples = 0 – 62500, compression = 25:1

→ 1000000 Samples

  - T(total) = **0,609s**, T(DPA) = 0,449s

- Least traces:
  - Traces = **157**, Samples = 10000 – 55000, compression = 25:1

→ 282600 Samples

  - T(total) = **0,278s**, T(DPA) = 0,232s

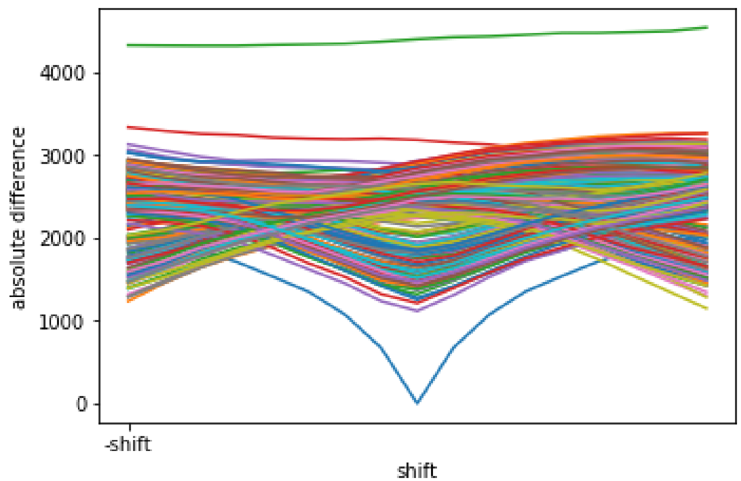
- Least time:
  - Traces = 380, Samples = 32000– 53000, compression = **500:1**

→ 15960 Samples

  - T(total) = **0,077s**, T(DPA) = 0,030s

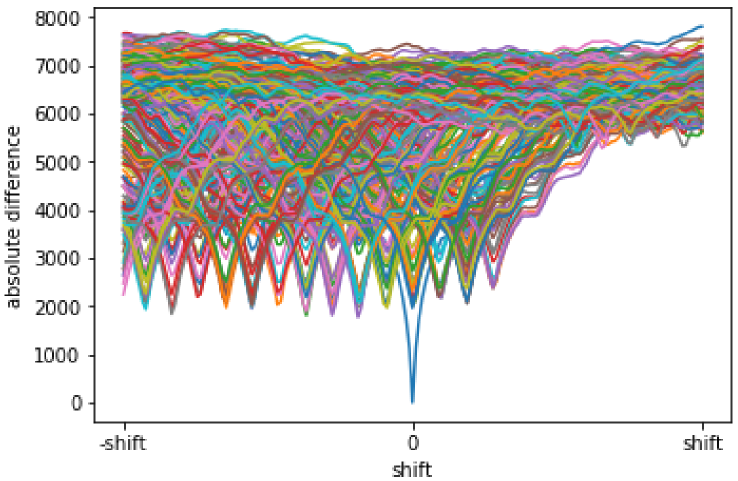
Benchmarks: Hiding

Least time for 3 nops



- t(total) = **1,312s** , t(DPA) = 0,176s
- Traces = 280, alignment = 0,015%
- Samples = 28000 – 55000

Least time for 15 nops



- t(total) = **11,788s**, t(DPA) = 0,269s
- Traces =270, alignment = 0,09%
- Samples = 0 - 50000

Benchmarks: Shuffling

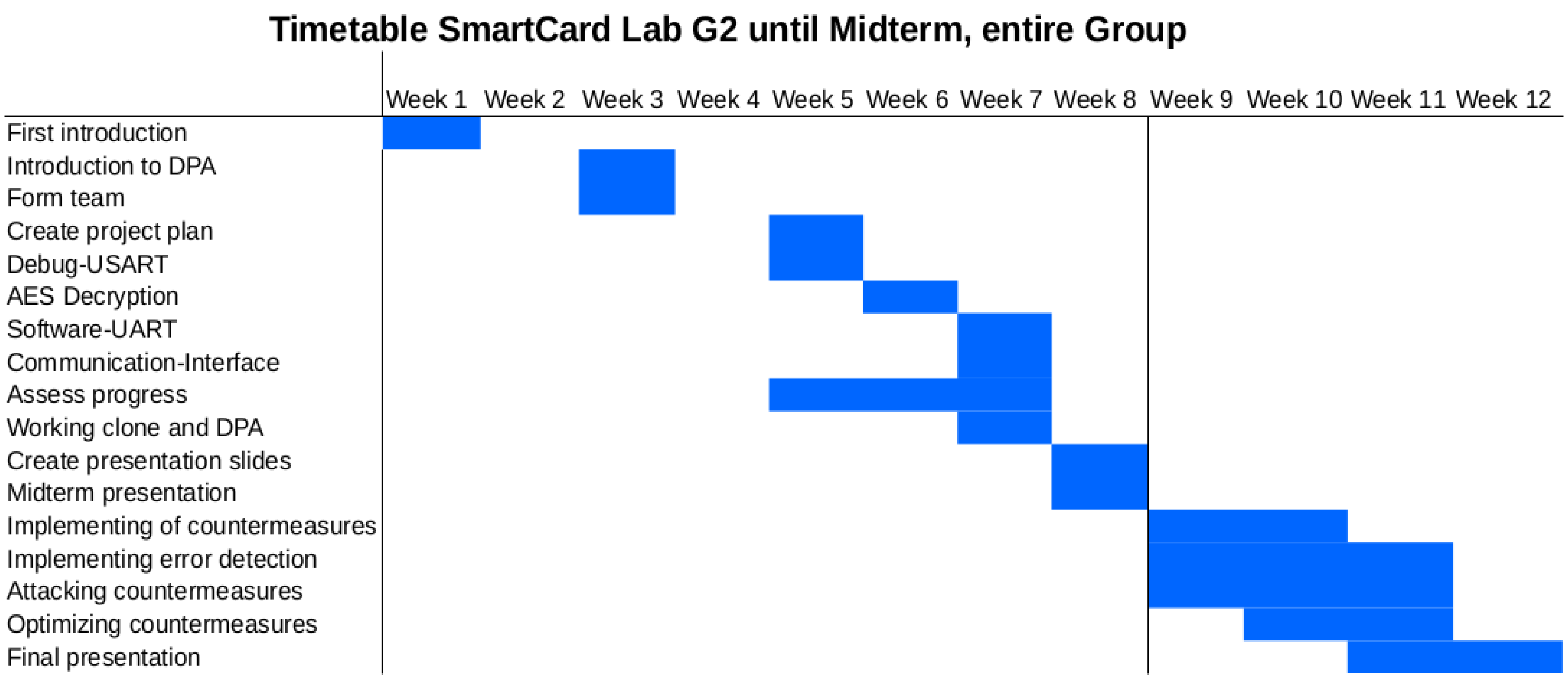
- We broke the first version of shuffling very easily
- Second implementation needed more tinkering:
  - Traces = 10000, Samples 100000 – 118900, compression = 27:1
  - Trace alignment and running mean (100)
  - **T(total) =51,426s**, T(DPA)=1,846s

Benchmarks: Masking

- Second-Order DPA works on unprotected implementation
  - Traces = 180, Samples = 12000 – 50000, reduction = 100
  - T(total) = **1,920s**

- Second-Order DPA on masking fails
  - Traces = 10000, Samples = 0 – 62500, reduction = 25
  - T(total) = **~6 min**

Project Plan



Thank you for your attention!