Chair of Security in Information and Technology
School of Computation, Information and Technology
Technical University of Munich

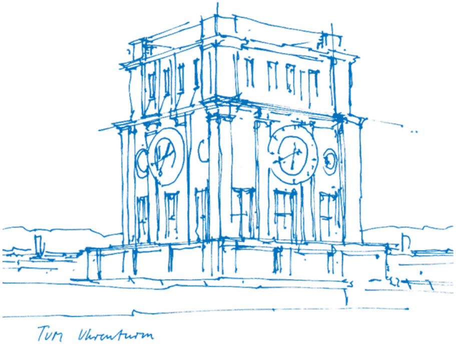
SmartCard Lab: Final Presentation

Group 2

Mevlüt Yıldırım & Sait Sevban Cander

Chair of Security in Information and Technology
School of Computation, Information and Technology
Technical University of Munich

January 22th, 2025



Outline

Task Distribution Among Team Members

Mevlüt Yıldırım

- Implementation of Random Number Generator
- Implementation of AES Countermeasures
- Countermeasure Attacks

Sait Sevban Cander

- Countermeasure Attacks
- Conducting DPA attack benchmarks against countermeasure

Outline

True Random Number Generator (TRNG)

Common entropy sources in STM32 microcontrollers include:

- Measuring jitter between two independent clocks:
  - SysTick timer combined with RTC/Watchdog (WDG).
- Utilizing the noisy internal temperature sensor value of the ADC.

For this implementation, the ADC was chosen as the entropy source for the TRNG.

ADC Setup & Read

ADC Setup:

- Short sampling time (13.5 cycles) for maximum jitter.
- 12-bit resolution.
- Temperature channel selected as the entropy source.

```
adc_enable_temperature_sensor();
adc_set_sample_time_on_all_channels(ADC1, ADC_SMPTIME_013DOT5);
adc_set_resolution(ADC1, ADC_RESOLUTION_12BIT);
adc_set_regular_sequence(ADC1, 1, channel_array2);
```

```
#define CHANNEL_ARRAY2 {1, 1, ADC_CHANNEL_TEMP}
```

ADC Read:

- Extract the least significant bit (LSB) from each conversion and apply an XOR-based correction to enhance randomness.

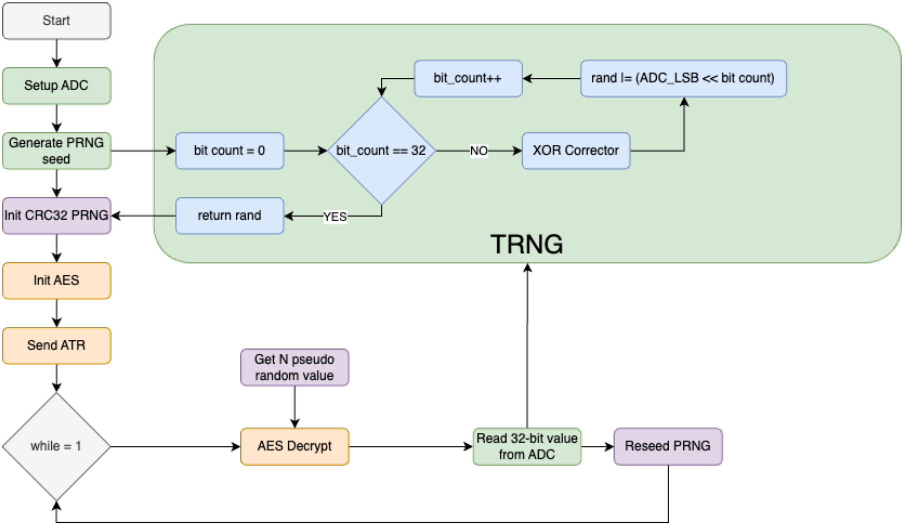
```
// Collect enough valid bits to form a 32-bit random
while (bit_position < 32)
{
    adc_start_conversion_regular(ADC1);
    while (!(adc_eoc(ADC1)))
        ;                                                /
    uint32_t adc_value_1 = adc_read_regular(ADC1); /
    uint32_t lsb_value_1 = adc_value_1 & 0x1;      /
    adc_start_conversion_regular(ADC1);
    while (!(adc_eoc(ADC1)))
        ;                                                /
    uint32_t adc_value_2 = adc_read_regular(ADC1); /
    uint32_t lsb_value_2 = adc_value_2 & 0x1;      /
    // XOR the two LSBs and add to the result
    uint32_t xor_result = lsb_value_1 ^ lsb_value_2;
    rand_value |= (xor_result << bit_position); // A
    bit_position++;
    // delay_cyc(10); // Allow time for jitter
}
return rand_value;
```

Pseudo-Random Number Generator (PRNG)

Overview:

- Cryptographically secure SHA-256 caused delays and timeouts during video stream decryption.
- CRC32 was chosen for its speed and efficiency, despite lacking cryptographic security.
  - Compatible with 32-bit entropy
  - Lightweight and fast
  - Easy to implement and use.
- Base implementation was sourced from GitHub and integrated into the OS (STM32's built-in CRC peripherals were a possible alternative).
- Reseeding after every decryption was used to improve security.

RNG Architecture



Outline

Shuffling Countermeasure

Permutation Generation:

- A 16-element permutation was generated using the PRNG and Fisher-Yates algorithm.
- Each element represents the index of data in the AES state.

```c
for (uint8_t i = len - 1; i > 0; i--)
{
    uint32_t rand = PRNG_Generate();
    uint8_t j = rand % (i + 1);
    // Swap perm[i] and perm[j]
    uint8_t temp = perm[i];
    perm[i] = perm[j];
    perm[j] = temp;
}
```

Shuffled AES Operations:

- The permutation was used to randomize operations in AES stages.
- Applied to AddRoundKey and InvSubBytes.

```c
static void ShuffledInvSubBytes(state_t *state, uint8_t *perm)
{
  for (uint8_t idx = 0; idx < 16; ++idx)
  {
    // Apply the permutation to access state and round key
    (*state)[perm[idx] / 4][perm[idx] % 4] = getSBoxInvert((*state)[perm[idx] / 4][perm[idx] % 4]);
  }
}
```

```c
static void ShuffledAddRoundKey(uint8_t round, state_t *state, const uint8_t *RoundKey, uint8_t *perm)
{
  uint8_t idx;
  for (idx = 0; idx < 16; ++idx)
  {
    // Apply the permutation to access state and round key
    (*state)[perm[idx] / 4][perm[idx] % 4] ^= RoundKey[(round * Nb * 4) + perm[idx]];
  }
}
```

Dummy Operations Countermeasure

- **Fixed Number of Operations:** 100 dummy operations are added per AES decryption.

Mimic S-Box:
Mimics the S-Box operation to obscure power traces.

```
for (uint8_t i = 0; i < iter_count; i++)
{
  uint8_t random_byte = PRNG_Generate() & 0xFF;
  uint8_t dummy_result = getSBoxInvert(random_byte);

  final_dummy_result ^= dummy_result;
  completed_dumops++;
}
```

Random Distribution:
Distributed randomly across AES rounds.

```
// Randomly distribute dummy operations across rounds
uint8_t dummy_ops_per_round[Nr];
memset(dummy_ops_per_round, 0, sizeof(dummy_ops_per_round));

for (uint8_t i = 0; i < Nr; i++)
{
  // Allocate a random portion of the remaining dummy operations
  if (remaining_dummyops > 0)
  {
    uint8_t allocation = PRNG_Generate() % (remaining_dummyops + 1);
    dummy_ops_per_round[i] = allocation;
    remaining_dummyops -= allocation;
  }
}
```

Random Order:
Inserted with a 50% chance before or after actual operations.

```
if (PRNG_Generate() % 2 == 0)
{
  InvSubBytes(state);
  DummyInvSubBytes(dummy_ops_per_round[round]);
}
else
{
  DummyInvSubBytes(dummy_ops_per_round[round]);
  InvSubBytes(state);
}
```

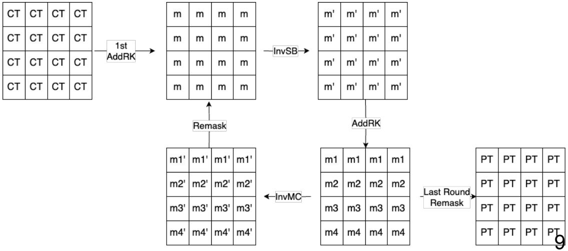
- Results are stored in volatile variables to prevent compiler optimizations.

Masking Countermeasure

- Total of 10 masks: 6 are random ($m1$, $m2$, $m3$, $m4$, m, and m') and 4 are computed ($m1'$, $m2'$, $m3'$, $m4'$).
- m and m' are used as input and output masks in the **InvSubBytes** operation.
- $m1$-$m4$ and $m1'$-$m4'$ are used as input and output masks in the **InvMixColumns** operation.
- **Decryption Flow:** Mask Init $\rightarrow$ Masked AddRK $\rightarrow$ [InvSR $\rightarrow$ Masked InvSB $\rightarrow$ Masked AddRK $\rightarrow$ (InvMC $\rightarrow$ ReMask)] $\rightarrow$ Final ReMask.

Mask Init:

- Precompute the masked invSB.
- Mask the final round key with m.
- Mask all other round keys with $m1 - m4$ & m'.



OS and AES Results

AES Execution Time

Implementation	Time [ms]	Difference [%]
Reference card (no countermeasure)	54.1	N/A
Own OS No Countermeasure	51.5	-4.8%
Own OS Dummy Ops.	53.9	+4.66%
Own OS Shuffling	51.8	+0.58%
Own OS Masking	53.7	+4.27%
Own OS All 3	56.6	+9.90%

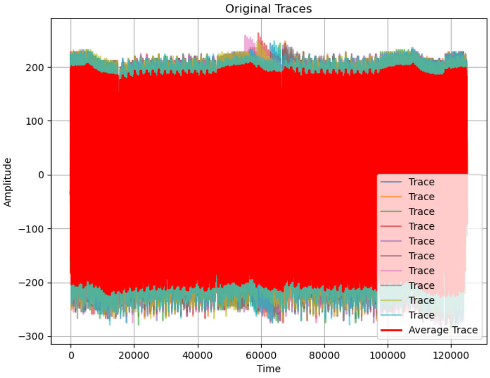
Memory Usage

Implementation	Text [kB]	Data [B]	BSS [B]	Total [B]	Difference [%]
Own OS No Countermeasure	3.8	8	176	3952	N/A
Own OS Dummy Ops.	5.856	28	192	6036	+52.77
Own OS Shuffling	5.716	28	192	5972	+51.15
Own OS Masking	6372	28	448	6848	+173.27
Own OS All 3	6660	28	452	7140	+180.66

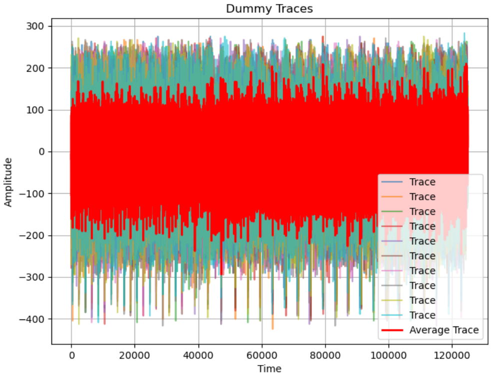
Outline

Attacking Own Card

No protection

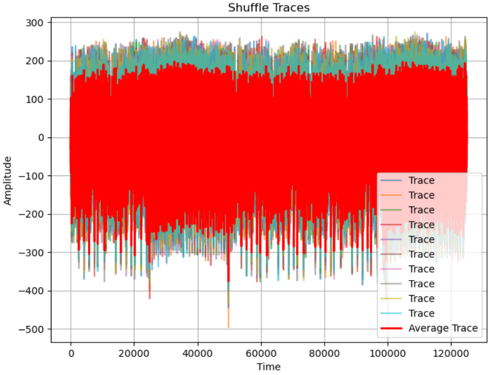


Dummy Operations

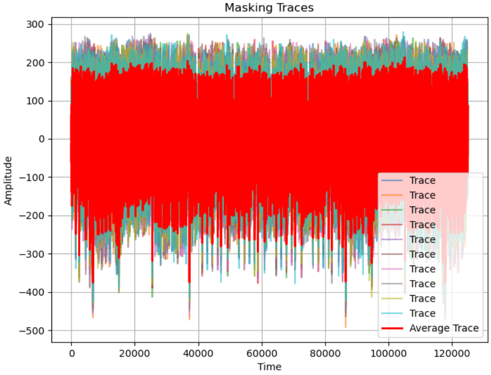


Attacking Own Card

Shuffling



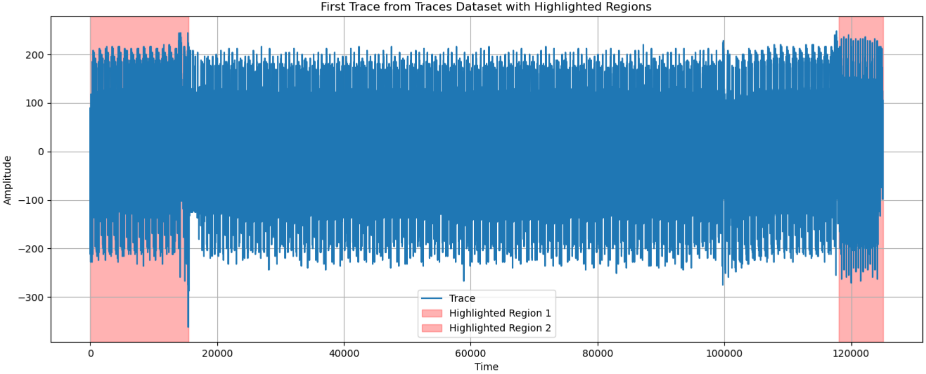
Masking



Possible Reasons

- Too many traces are required
- Problems due to misalignment
- Need of a more advanced attack (2nd order)
- Sample count uncertainty

Attack on Reference Card + Shuffling



DPA Attack Benchmark Results

Implementation	Broken [yes/no]	Min. # of Traces	Duration [s]	Compression Method	Window Size
Own Card (no countermeasure)	Yes	250	0.91	Squared	7
Own Card + Dummy Operations	X	X	X	X	X
Own Card + Shuffling	Yes	400	0.65	Absolute	13
Own Card + Masking	X	X	X	X	X
Reference Card (no countermeasure)	Yes	200	0.86	Max	9
Reference Card + Dummy Operations	X	X	X	X	X
Reference Card + Shuffling	Yes	7050	0.7	Absolute	17
Reference Card + Masking	X	X	X	X	X