PP Smartcard: Final Presentation

Team 1

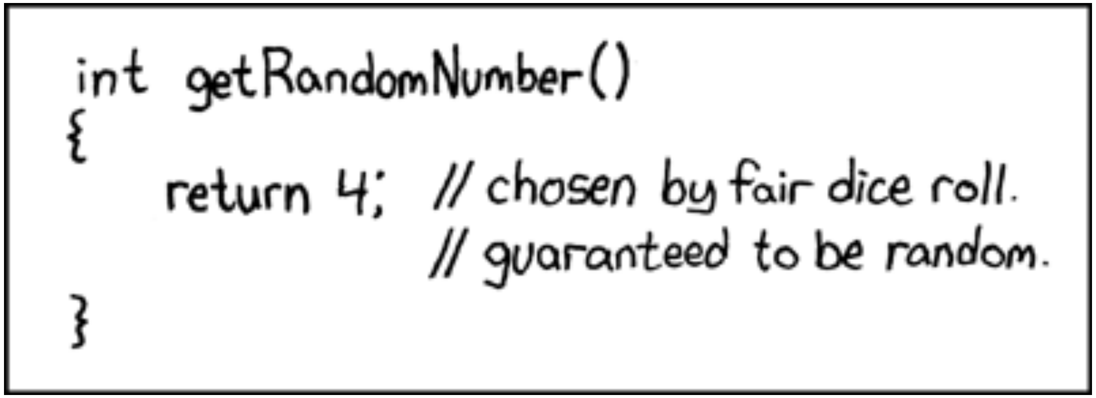
Laurenz Altenmüller
Carlo van Driesten
Markus Hofbauer
Kevin Meyer
Johannes Schreiner

14.07.2015

Agenda

1. Random Number Generation
2. Countermeasures
   a. Own Implementation
   b. Dummy Operations
   c. Shuffling
   d. Masking
3. Attack
   a. Countermeasure Attacks
   b. DPA Improvements
   c. Masking
4. Conclusion

Random Number Generator



https://xkcd.com/221/

Random Number Generation

RNG possible with

(1) HW random number generation sources
(2) cryptographically secure pseudo random number generator and a small amount of random entropy (e.g. 128 bits)

(1) Atmega644 does not have a HW RNG
→ divert other hardware features to harvest randomness
General Problem: high bitrate hard to achieve

our approach

(2) Harvest low bitrate entropy from Atmega644 hardware features and generate high bitrate pseudorandom numbers with cryptographic algorithm

Hardware RNG on Atmega644

Atmega644 does not have a HW RNG
→ Harness other hardware features for randomness harvesting

our approach

ADC

Floating ADC pins, lower bits of digitized values change unpredictable

+ High bit change rate → high entropy bitrate

- Worthless for us as attacker can simply pull the pins to zero

Uninitialized Memory

Some bits always 0 or 1
Some "randomly" differ after reboots

+ Easy to harvest (XOR every byte in uninitialized block of memory)

- Entropy needs to be collected before RAM is used
- Finite amount per execution

Watchdog Timer

Jitter between internal watchdog oscillator and smartcard terminal clock

+ Sources known: depends on manufacturing **and** environment

- Low bitrate

RNG based on Watchdog Timer Jitter

Watchdog interrupt: Triggered by watchdog timer (~ every 16ms)
Uses internal watchdog clock with high jitter

Entropy collection: Counter TCNT0 running in *normal mode*
Every WDT ISR stores 8-bit counter value:

`0x11XXXXXX`

identical most of the time | appear to be "random"

After 32 interrupts (512ms): 256 bit of counter values,
128 appear to be random

→ create one 32 bit value with jenkins hash (shift and XOR)

Peak performance: ~64 bit/second

Problems:

- Good quality, but: bitrate too low for RNG tests DIEHARD (requires > 300MB) and DIEHARDER (> 1GB)
- Bitrate could be higher for our applications
- First byte after 0.5 seconds, protocol allows AES before!

Cryptographically Secure Pseudo RNG

- PRNG based on Skein hashing (as specified in Skein specification)
- Skein: SHA-3 competition finalist, based on Threefish block cipher
- Can be used with 128 bit entropy

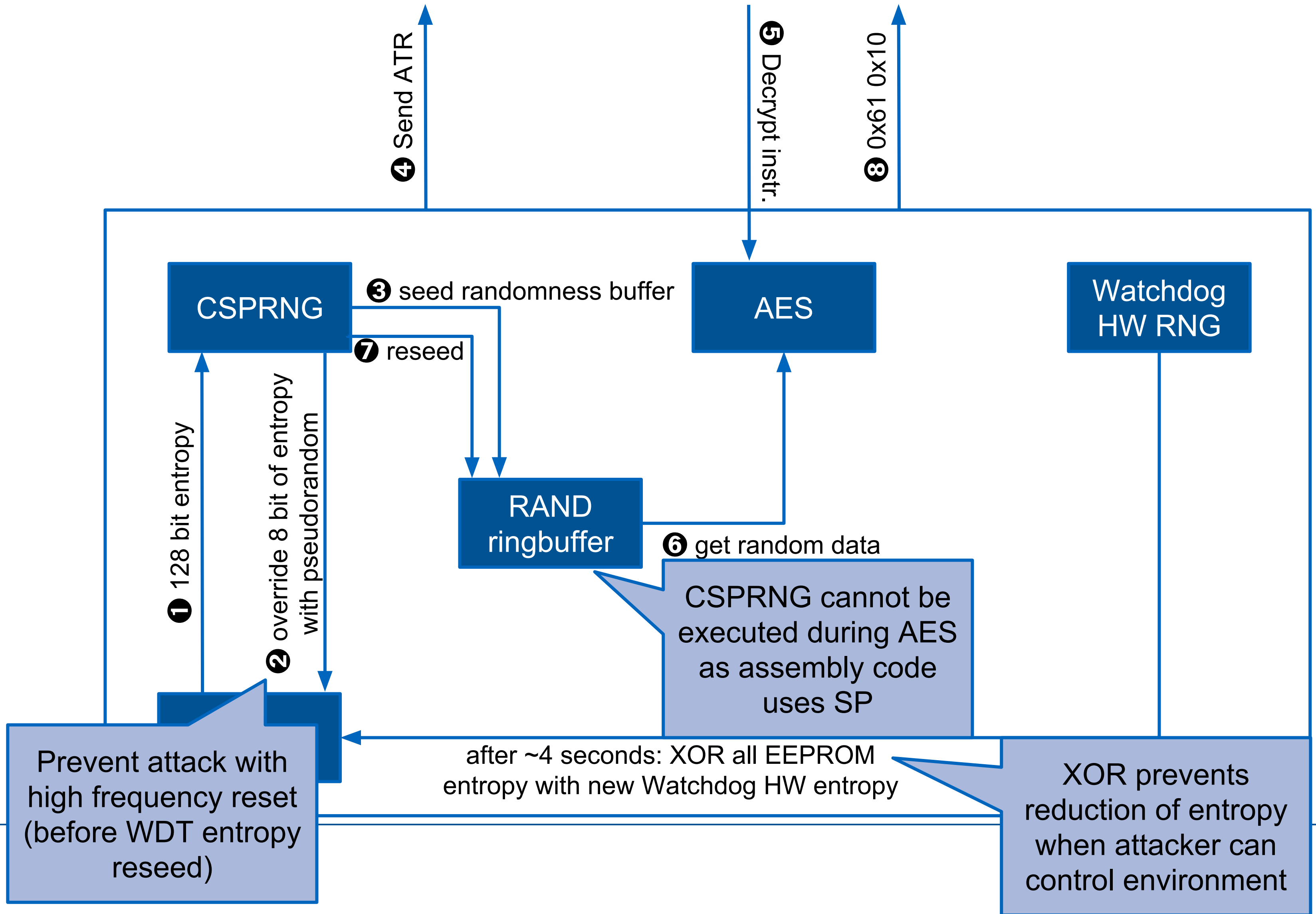
Implementation: Fhreefish library, targeted at high performance applications

- Assembly implementation
- **Overall performance**: 285 cycles per byte

Decision process:

- Fhreefish is the best performance implementation of a CSPRNG for AVR
- Cryptographic quality by far high enough
  → Entropy gathering remains weakest point in our RNG architecture

RNG Architecture



RNG Architecture - Performance & Security

- Terminal accepts more than 250 ms delay for encryption
  → use time after encryption to refill RNG buffer
- Sufficient for generation of more than 1000 bytes of pseudorandom
  → more than enough for any countermeasure

Security considerations:

- Quality of entropy not really known
- New WDT entropy is added to EEPROM via XOR
  → entropy quality cannot decrease
- EEPROM values could also be initialized to truly random values after programming of card

⇒ Attacking RNG is not the easiest way to attack the hardened card!

PRNG Analysis

https://www.random.org/analysis/dilbert.jpg

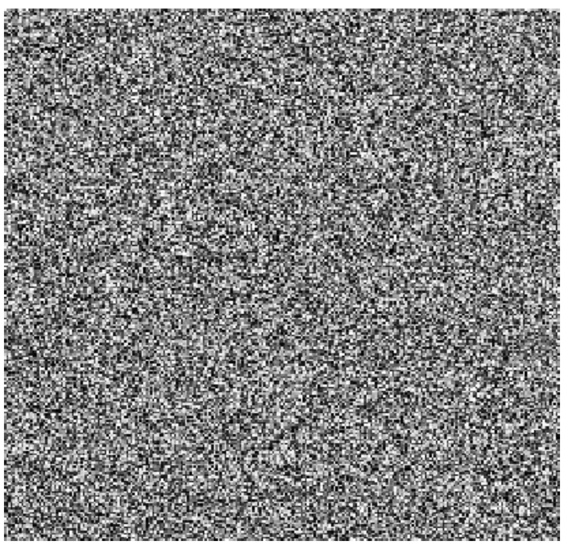
PRNG Analysis

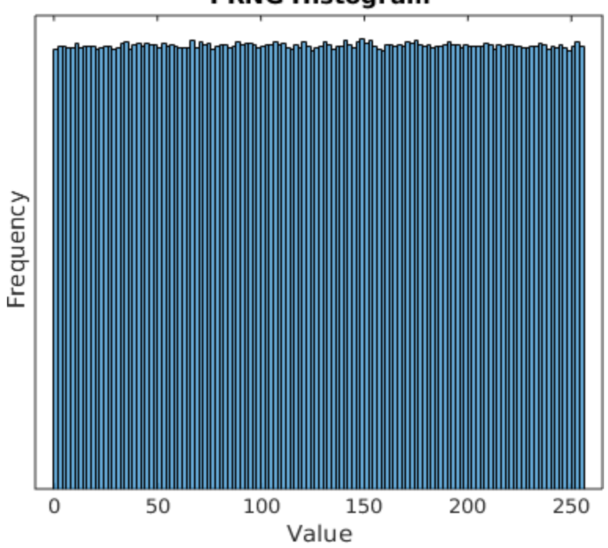
		Compression,	Entropy
•	Ours:	100.0166%	7.999986
•	C rand()	100.0167%	7.994336

```
>> runstest(x)

ans =

     0
```





Agenda

1. Random Number Generation
2. Countermeasures
   a. Own Implementation
   b. Dummy Operations
   c. Shuffling
   d. Masking
3. Attack
   a. Countermeasure Attacks
   b. DPA Improvements
   c. Masking
4. Conclusion

Own Implementation (-Os)

	Original	**Clone** *(improved)*
Total code size	*unknown*	4850 B
Total data size	*unknown*	253 B
AES execution time	**4.604 ms**	**4.087 ms** (4.5 ms)
AES code size	*unknown*	1816 B
AES data size	*unknown*	192 B

Own Implementation

	O0	**O3**	**Os**
Traces (min)	200	500	1000
Time	0.76s	1.54s	3.06s

- Compiler optimization influences the code structure
- **util/delay.h** relies on optimization flags

Dummy Operations

- Dummy Operations = Table Lookups
  → Disguised as SB operation

- Used at:
  - First round SB
  - Last round SB
  - End of AES (remaining)

- Current configuration (per AES):
  - 80 operations total
  - 240 random values
  - 3 Compares / 1 modulo

```
// get a random value for the XOR operations
uint8_t value = rng_get_random_byte();

// this loop will at least execute once
// operations = rng_get_random_byte() % (modulo + 1);
do {
    // use arbitrary index for table lookup
    uint8_t rndIndex = rng_get_random_byte();

    // dummy operation should be similar to the
    // original SB operation
    value ^= pgm_read_byte(aes_invsbox+rndIndex);

    ++completedOperations;
    ++counter;
} while (counter < operations);

// XOR on volatile to avoid compiler optimization
dummy_result ^= value;
```

- Same amount of operations per AES cycle!!!
  (no information retrievable from execution times)

Dummy Operations (cont.)

	Base	Base + RNG	Dummy Operations
Total code size	4,850B (7.4%)	12,222B (18.6%)	12,438B (19.0%)
Total data size	253B (6.2%)	1,550B (37.8%)	1,554B (37.9)
AES execution time	**4.09 ms**	**4.09 ms**	**13.25 ms**

Shuffling

- Possibilities:
  - Random array permutation (Fisher–Yates shuffle)
  - Operations at SB and RK in arbitrary order
  - SB and SR interchangeable

```
/**************************************************/
/***              Permutation Array             ***/
/**************************************************/
void aes128_update_shuffling_array() {

    uint8_t i = 15; // array end
    uint8_t j = 0;  // init value
    do{
        //generate a random number [0, n-1]
        j = rng_get_random_byte() % (i+1);

        //swap the last element with element at random index
        uint8_t temp = permutation_array[i];
        permutation_array[i] = permutation_array[j];
        permutation_array[j] = temp;
        --i;
    } while ( i != 0);
}
```

- Used at:
  - All RK operations
  - All SB operations
  - 2 times array refresh per AES

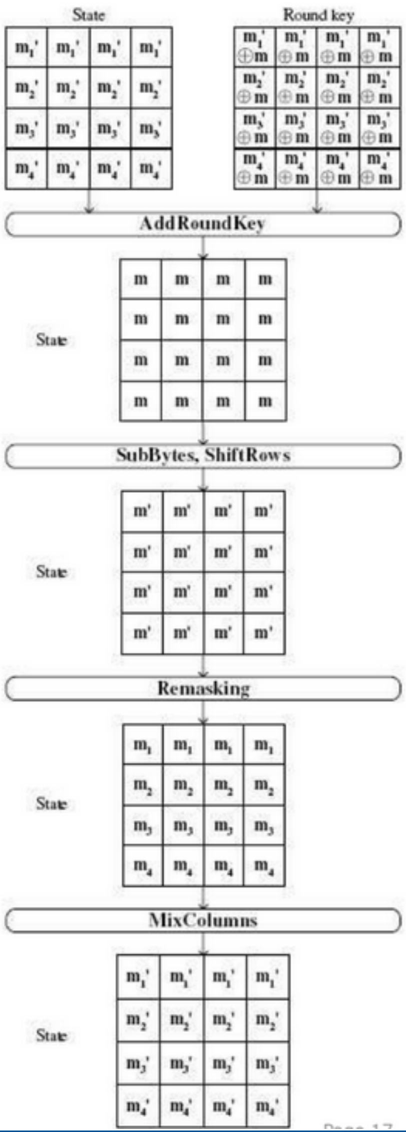
- Current configuration (per AES):
  - 2 * 15 = 30 random numbers
  - 2 * 15 * 3 = 90 copy operations

Shuffling (cont.)

	Base + RNG	**Shuffling**
Total code size	12,222B (18.6%)	12,446B (19.0%)
Total data size	1,550B (37.8%)	1,566B (38.2%)
AES execution time	**4.087 ms**	**7.40 ms (+3.31 ms)**

Masking

- Precompute masks:
  - 6 random masks, 4 computed masks
  - Precompute inverse SBox values
- Mask expansion key:
  - Expansion key calculation before the ATR
  - Masking of expansion key before each AES
- Start: Mask challenge
- End: Unmask challenge
- Remasking in between: MK → possible mask removal
- Mask State / Expansion Key -> SR -> SB
  -> [RK -> MK -> invremask -> SR -> SB] -> RK -> unmask



Masking (cont.)

	Base + RNG	**Masking**
Total code size	12,222B (18.6%)	13,080B (20.0%)
Total data size	1,550B (37.8%)	1,992B (48.6%)
AES execution time	**4.087 ms**	**6.4 ms (+ 2.61 ms)**

- New SBox uses a lot of data space
- In-place calculation of SBox possible and tested
  (no extra memory, but longer computation time)

Agenda

1. Random Number Generation

2. Countermeasures
   a. Own Implementation
   b. Dummy Operations
   c. Shuffling
   d. Masking

3. **Attack**
   a. Countermeasure Attacks
   b. DPA Improvements
   c. Masking

4. Conclusion

Countermeasures Attack

	Traces	Time	% rel. to Pure AES
Original	200	0.63s	40%
Pure AES	500	1.54s	100%
Dummy Ops.	3200	9.86s	+540%
Shuffling	6300	18.63s	+1160%

- DPA Improvements:
  - compression more efficient
  - core dpa works good with countermeasure

Second Order DPA (1)

Idea:

- requirement: two intermediate values, sharing the same mask
- $u_m \oplus v_m = (u \oplus m) \oplus (v \oplus m) = u \oplus v$

Preprocessing is needed in order to attack:

- $|HW(u_m) - HW(v_m)|$
- new samples size $\tilde{l} = \dfrac{l \cdot (l-1)}{2}$

DPA:

- hypothetical power consumption:
  $H = HW(u \oplus v)$
- same correlation core

Second Order DPA (2)

Results:

- first-order DPA: **not successful**
  $\Rightarrow$ Implementation correct
- second-order DPA: **not successful**

Problem:

- our masking implementation doesn't reuse masks
  $$m \neq m'$$

$\Rightarrow$ not vulnerable by second-order DPA

Practical Problems:

- huge amount of traces
- heavy compression required to get reasonable amount of samples
- quadratic effort due to preprocessing

Agenda

Project Management

Problems:

- RNG
  - complex implementation
  - blocking task for countermeasures

Plan → Reality ✔

Conclusion

- Learned:
  - In-depth & hands-on SmartCard & DPA knowledge
  - Algorithmic efficiency
  - Don’t rely on security implemented by others
  - Review sessions
- Achieved:
  - Attack on original card
  - Cloned the SmartCard
  - Hardened the CloneCard
  - Good randomness solution
  - Attack-Analysis on hardened card
  - Great team & teamwork
- Further improvements:
  - Faster AES implementation
  - Examine optimal compiler flags
  - Combine hiding techniques

Thank you!