Speed and Size-Optimized Implementations of the PRESENT Cipher for Tiny AVR Devices

Kostas Papagiannopoulos
Aram Verstegen

December 14, 2013

Who We Are

- 2-year Master's programme in computer security
- Collaboration of 3 universities
- Software, Hardware, Networks, Formal methods, Cryptography, Privacy, Law, Ethics, Auditing, Physics
- http://kerckhoffs-institute.org/

Lightweight ciphers

- What is the main functional requirement of a crypto primitive?

Lightweight ciphers

- What is the main functional requirement of a crypto primitive?
  - Security

Lightweight ciphers

- What is the main functional requirement of a crypto primitive?
  - Security

Lightweight ciphers

- What is the main functional requirement of a crypto primitive?
  - Security

- Is it the only one?

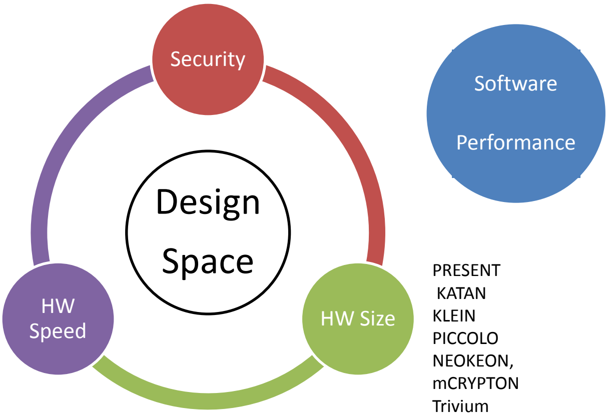
Lightweight ciphers

- What is the main functional requirement of a crypto primitive?
  - Security

- Is it the only one?
  - Ubiquitous computing led to hardware size issues
  - Is AES good enough?
  - The story of Keeloq

Lightweight ciphers

- Software performance? Throughput, latency?



PRINCE

PRESENT
KATAN
KLEIN
PICCOLO
NEOKEON,
mCRYPTON
Trivium
SPONGENT

ATtiny Family

Model	Flash (Bytes)	SRAM (Bytes)	Clock speed (MHz)
ATtiny13	1024	64	20
ATtiny25	2048	128	20
ATtiny45	4096	256	20
ATtiny85	8192	512	20
ATtiny1634	16384	1024	12

- Basic 90 (single word) AVR instructions
- 32 8-bit general purpose registers
- 16-bit address space
- 16-bit words
- Harvard architecture

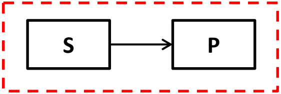
ATtiny45 Address Space

7 0	Addr.	16-bit	Use
R0	0x00		
R1	0x01		
R2	0x02		
..			
R13	0x0D		
R14	0x0E		
R15	0x0F		
R16	0x10		
R17	0x11		
..			
R26	0x1A	X low	SRAM
R27	0x1B	X high	
R28	0x1C	Y low	SRAM + CPU registers
R29	0x1D	Y high	
R30	0x1E	Z low	SRAM + Flash
R31	0x1F	Z high	
64 I/O registers	0x0020 - 0x005F		
Internal SRAM	0x0060 - 0x00DF		

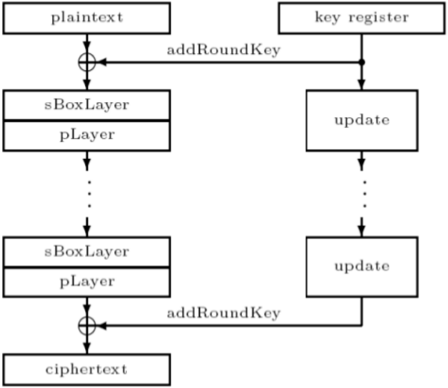
Quick AVR Recap

Load register from immediate	**ldi** *Rd*, *42*
Load register from SRAM pointer (X)	**ld** *Rd*, *X*
Load register from Flash pointer (Z)	**lpm** *Rd*, *Z*
XOR output with input	**eor** *Ro*, *Ri*
Swap nibbles in byte	**swap** *Rd*
Rotate left with carry	**rol** *Rd*
Rotate left without carry	**lsl** *Rd*
Store to SRAM from register (and increment)	**st** *X+*, *Rd*
Procedure calls	**rcall**, **ret**, **rjmp**
Stack access	**push**, **pop**
Counting	**inc**, **dec**
Adding	**add**, **sub**
Binary logic	**and**, **or**, **eor**

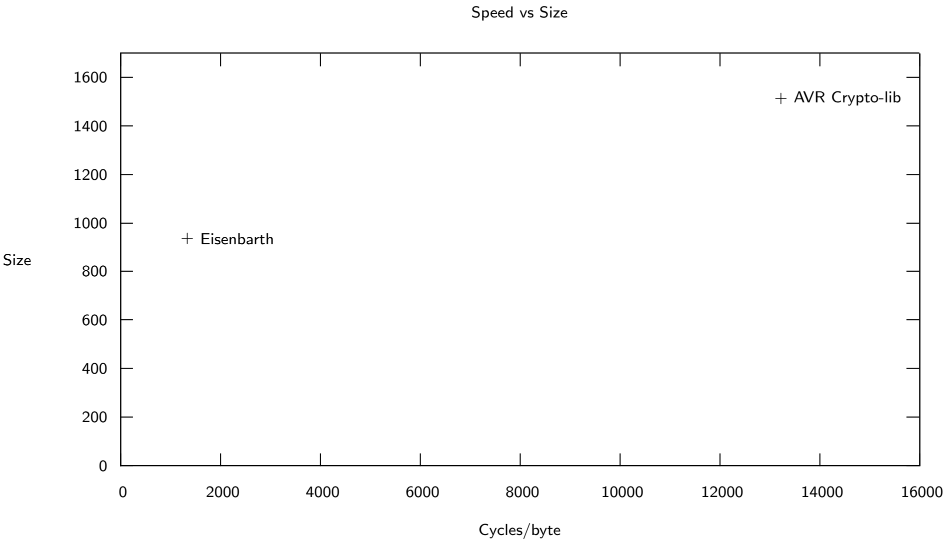
PRESENT cipher



- Substitution-Permutation network
- 31 rounds
- 64-bit block, 80-bit key
- XOR operation
- S-layer
- P-layer



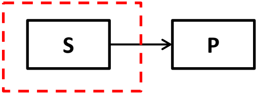
State of the Art



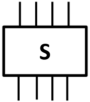
addRoundKey

```
; state ^= roundkey (first 8 bytes of key register)
addRoundKey:
    eor STATE0, KEY0
    eor STATE1, KEY1
    eor STATE2, KEY2
    eor STATE3, KEY3
    eor STATE4, KEY4
    eor STATE5, KEY5
    eor STATE6, KEY6
    eor STATE7, KEY7
    ret
```

PRESENT cipher

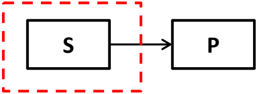


- We use a $GF(2^4) \rightarrow GF(2^4)$ S-box to generate non-linearity

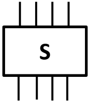


x	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
S[x]	C	5	6	B	9	0	A	D	3	E	F	8	4	7	1	2

PRESENT cipher



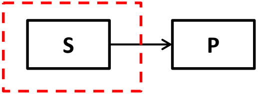
- We use a $GF(2^4) \rightarrow GF(2^4)$ S-box to generate non-linearity



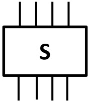
x	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
S[x]	C	5	6	B	9	0	A	D	3	E	F	8	4	7	1	2

- How is the S-box implemented?

PRESENT cipher



- We use a $GF(2^4) \rightarrow GF(2^4)$ S-box to generate non-linearity



x	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
S[x]	C	5	6	B	9	0	A	D	3	E	F	8	4	7	1	2

- How is the S-box implemented?
  - In software: lookup tables
  - In hardware: logic circuit based on the 4-to-4 Boolean function

4-bit S-Box

x	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
S[x]	C	5	6	B	9	0	A	D	3	E	F	8	4	7	1	2

4-bit S-Box

x	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
S[x]	C	5	6	B	9	0	A	D	3	E	F	8	4	7	1	2

- Accessing the table 4 bits at a time incurs a penalty

```
low_nibble:
  mov ZL, INPUT        ; load input
  andi ZL, 0xF         ; take low nibble as table index
  lpm OUTPUT, Z        ; load table output
  cbr INPUT, 0xF       ; clear low nibble
  and INPUT, OUTPUT    ; save low nibble to input
  ret
byte:
  rcall low_nibble     ; substitute low nibble
high_nibble:
  swap INPUT           ; swap nibbles
  rcall low_nibble     ; substitute low nibble
  swap INPUT           ; swap nibbles back
  ret
```

4-bit S-Box

x	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
S[x]	C	5	6	B	9	0	A	D	3	E	F	8	4	7	1	2

- Accessing the table 4 bits at a time incurs a penalty

```
low_nibble:
  mov ZL, INPUT       ; load input
  andi ZL, 0xF        ; take low nibble as table index
  lpm OUTPUT, Z       ; load table output
  cbr INPUT, 0xF      ; clear low nibble
  and INPUT, OUTPUT ; save low nibble to input
  ret
byte:
  rcall low_nibble  ; substitute low nibble
high_nibble:
  swap INPUT          ; swap nibbles
  rcall low_nibble  ; substitute low nibble
  swap INPUT          ; swap nibbles back
  ret
```

- We have an 8-bit architecture, so we want to access bytes!

Squared S-Box

x	00	01	02	03	...	0C	0D	0E	0F
S[x]	CC	C5	C6	CB	...	C4	C7	C1	C2
x	10	11	12	13	...	1C	1D	1E	1F
S[x]	5C	55	56	5B	...	54	57	51	52
⋮	⋮	⋮	⋮	⋮	...	⋮	⋮	⋮	⋮
x	F0	F1	F2	F3	...	FC	FD	FE	FF
S[x]	2C	25	26	2B	...	24	27	21	22

Squared S-Box

x	00	01	02	03	...	0C	0D	0E	0F
S[x]	CC	C5	C6	CB	...	C4	C7	C1	C2
x	10	11	12	13	...	1C	1D	1E	1F
S[x]	5C	55	56	5B	...	54	57	51	52
⋮	⋮	⋮	⋮	⋮	...	⋮	⋮	⋮	⋮
x	F0	F1	F2	F3	...	FC	FD	FE	FF
S[x]	2C	25	26	2B	...	24	27	21	22

- New S-Box is 256 bytes, $16 \cdot 16$ combinations of two nibbles

Squared S-Box

x	00	01	02	03	...	0C	0D	0E	0F
S[x]	CC	C5	C6	CB	...	C4	C7	C1	C2
x	10	11	12	13	...	1C	1D	1E	1F
S[x]	5C	55	56	5B	...	54	57	51	52
⋮	⋮	⋮	⋮	⋮	...	⋮	⋮	⋮	⋮
x	F0	F1	F2	F3	...	FC	FD	FE	FF
S[x]	2C	25	26	2B	...	24	27	21	22

- New S-Box is 256 bytes, $16 \cdot 16$ combinations of two nibbles
- It substitutes 1 byte at a time

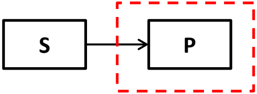
Squared S-Box

x	00	01	02	03	...	0C	0D	0E	0F
S[x]	CC	C5	C6	CB	...	C4	C7	C1	C2
x	10	11	12	13	...	1C	1D	1E	1F
S[x]	5C	55	56	5B	...	54	57	51	52
⋮	⋮	⋮	⋮	⋮	...	⋮	⋮	⋮	⋮
x	F0	F1	F2	F3	...	FC	FD	FE	FF
S[x]	2C	25	26	2B	...	24	27	21	22

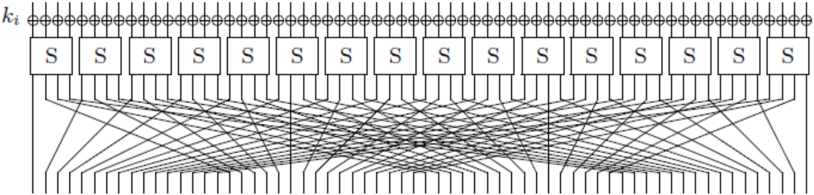
- New S-Box is 256 bytes, $16 \cdot 16$ combinations of two nibbles
- It substitutes 1 byte at a time
- No need to swap or discern high/low nibble

```
mov ZL, INPUT ; load table input
lpm OUTPUT, Z ; save table output
ret
```

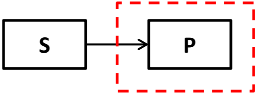
PRESENT cipher



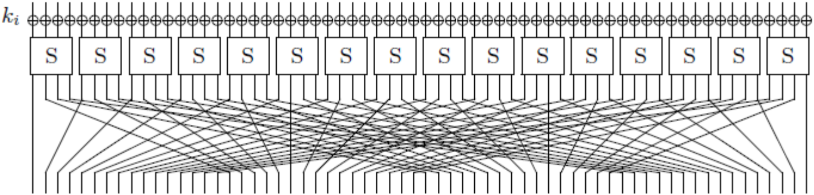
- Diffusion is achieved via pattern of bit-level permutations



PRESENT cipher

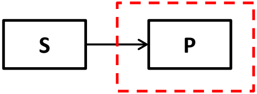


- Diffusion is achieved via pattern of bit-level permutations

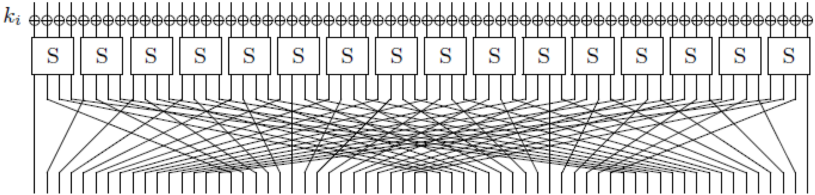


- How is the P-layer constructed?

PRESENT cipher

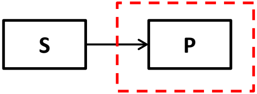


- Diffusion is achieved via pattern of bit-level permutations

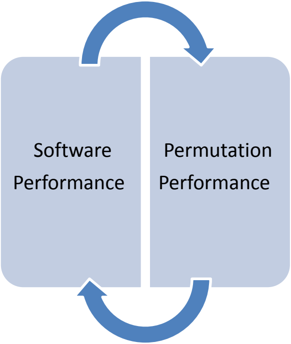


- How is the P-layer constructed?
  - In hardware: wires!
  - In software: mask and shift? lookup tables? Something else?

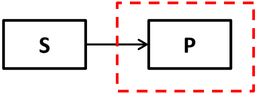
PRESENT cipher



- Software performance relies heavily on the permutation layer
- We need to focus on efficient permutation methods

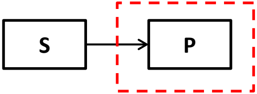


PRESENT cipher



- Implement with a masking and shifting

PRESENT cipher



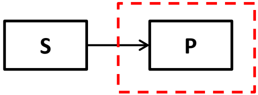
- Implement with a masking and shifting

```
1. Extract bit from a
     a7 a6 a5 a4 a3 a2 a1 a0
^     0  0  0  0  1  0  0  0
     0  0  0  0 a3  0  0  0

2. Shift to right position
    0 a3  0  0  0  0  0  0

3. Deposit bit to b
   b7  0 b5 b4 b3 b2 b1 b0
v   0 a3  0  0  0  0  0  0
   b7 a3 b5 b4 b3 b2 b1 b0
```

PRESENT cipher



- Implement with a masking and shifting

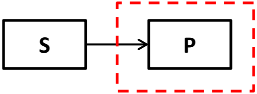
```
 1. Extract bit from a
     a7 a6 a5 a4 a3 a2 a1 a0
^     0  0  0  0  1  0  0  0
      0  0  0  0 a3  0  0  0

2. Shift to right position
    0 a3  0  0  0  0  0  0

3. Deposit bit to b
   b7  0 b5 b4 b3 b2 b1 b0
v   0 a3  0  0  0  0  0  0
   b7 a3 b5 b4 b3 b2 b1 b0
```

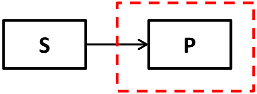
- Shifting cost is *proportional* to the argument
  - N shifts == N clock cycles

PRESENT cipher



- AVR is bad at shifting
- Issue was addressed with bit level instructions `bld, bst`

PRESENT cipher

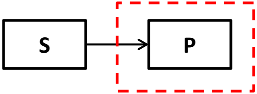


- AVR is bad at shifting
- Issue was addressed with bit level instructions `bld`, `bst`

a7 a6 a5 a4 **a3** a2 a1 a0

b7 **a3** b5 b4 b3 b2 b1 b0

PRESENT cipher



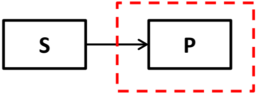
- AVR is bad at shifting
- Issue was addressed with bit level instructions `bld, bst`

```
a7 a6 a5 a4 a3 a2 a1 a0
```

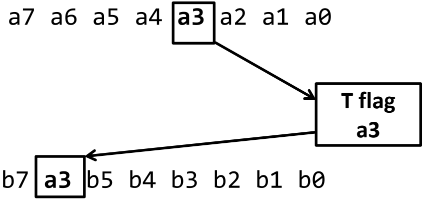
T flag
a3

```
b7 a3 b5 b4 b3 b2 b1 b0
```

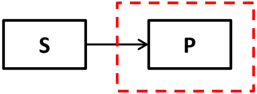
PRESENT cipher



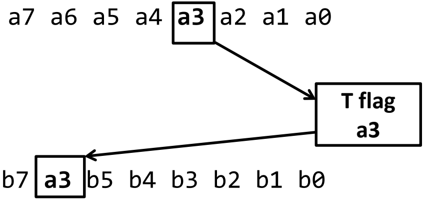
- AVR is bad at shifting
- Issue was addressed with bit level instructions `bld, bst`



PRESENT cipher

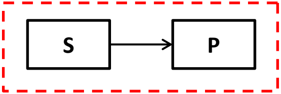


- AVR is bad at shifting
- Issue was addressed with bit level instructions `bld, bst`

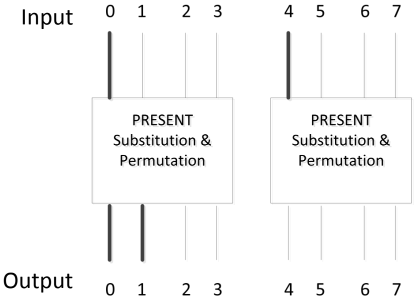


- Instructions `bld, bst` cost 1 cycle each, thus a 64-bit permutation can be done in 128 cycles [*2*n*]
- Still, with these techniques, you will get an implementation around 10-11k cycles [*Eisenbarth et al. ECRYPT*]

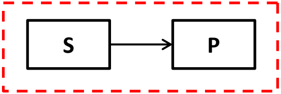
PRESENT cipher



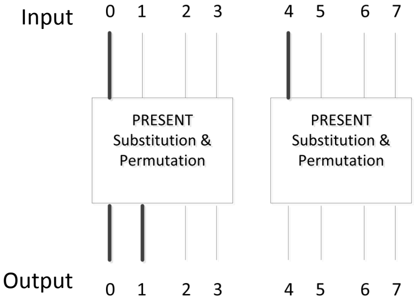
- Lookup tables for *merged* S and P layers



PRESENT cipher

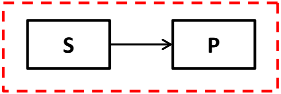


- Lookup tables for *merged* S and P layers

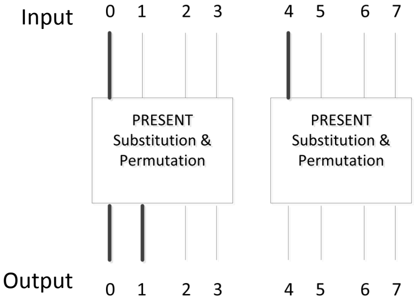


- A single input byte, contributes 2 bits to the output

PRESENT cipher

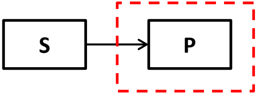


- Lookup tables for *merged* S and P layers



- A single input byte, contributes 2 bits to the output
- This technique will result in an implementation around 8.7k cc [*Papagiannopoulos, Verstegen RFIDsec*]

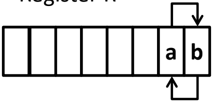
PRESENT cipher



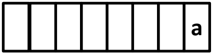
- A new representation is in order
- A Bitslicing tutorial

Permute bit a from position 1 to position 0 and bit b from position 0 to position 1

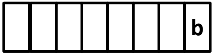
Register R



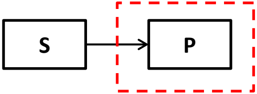
Register R



Register U

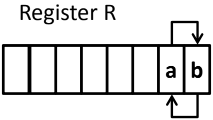


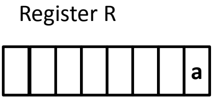
PRESENT cipher

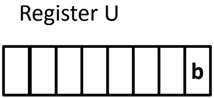


- A new representation is in order
- A Bitslicing tutorial

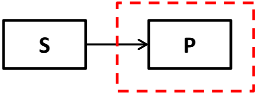
Permute bit a from position 1 to position 0 and bit b from position 0 to position 1





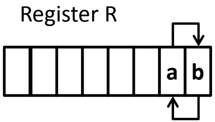


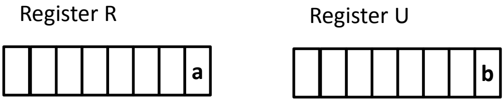
PRESENT cipher



- A new representation is in order
- A Bitslicing tutorial

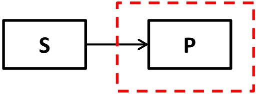
Permute bit a from position 1 to position 0 and bit b from position 0 to position 1





- Permutation computation is simple register re-naming, so ZERO cycles

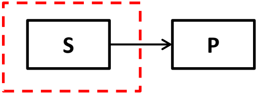
PRESENT cipher



- PRESENT P-layer contains lots of permutations
- We use bitslicing to compute the PRESENT P-layer
  - We don’t store a singe 64-bit state in 64/8 = 8 registers
  - We use 64 8-way bitsliced registers to represent 8 idependent cipher states

- Bitslicing essentially simulates the hardware behavior of wires in software
- Bitslicing offers us SIMD
  - Permuting a register results in 8 parallel bit permutations
  - Any operation (e.g. AND) between 2 bitsliced registers results in 8 parallel bit operations

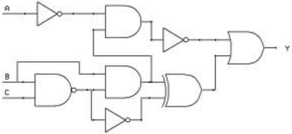
PRESENT cipher



- We have established the permutation layer implementation
- How will we compute the S-boxes under the bitsliced representation?
  - I can no longer use lookup tables!
  - Converting from/to bitsliced representation has severe overhead

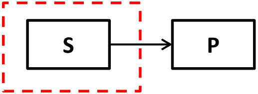
- I borrow a hardware method: Boolean function computation in software
  - The S-box can also be computed via an equivalent Boolean function

A	B	C	Y
0	0	0	
0	0	1	
0	1	0	
0	1	1	
1	0	0	
1	0	1	
1	1	0	
1	1	1	



- SIMD parallelization is maintained when computing a Boolean function
  - Every logical gate in software uses register operands, thus operates on 8 bits in parallel

PRESENT cipher



- How do we minimize the Boolean function?
- In general, NP-hard, even for 8-to-8 S-boxes!

- The Boyar-Peralta Heuristic
  - Start from a Boolean function using XOR,AND gates
  - Reduce the multiplicative complexity (number of AND operations)
  - Optimize linear components (XOR operations)

- This resulted in very efficient hardware AES implementations
- Does the same technique apply in software?

Key Update

1. Rotate 80-bit key register 61 bits to the left

Key Update

1. Rotate 80-bit key register 61 bits to the left
   - Rotate 19 bits to the right instead

Key Update

1. Rotate 80-bit key register 61 bits to the left
   - Rotate 19 bits to the right instead
   - Use 2 *mov* instructions to rotate $2 \cdot 8 = 16$ bits

Key Update

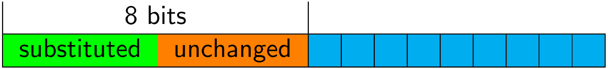
1. Rotate 80-bit key register 61 bits to the left
   - Rotate 19 bits to the right instead
   - Use 2 *mov* instructions to rotate $2 \cdot 8 = 16$ bits
   - Use *ror* only for the last 3 bits

Key Update

1. Rotate 80-bit key register 61 bits to the left
   - Rotate 19 bits to the right instead
   - Use 2 *mov* instructions to rotate $2 \cdot 8 = 16$ bits
   - Use *ror* only for the last 3 bits
2. S-Box the top 4 bits of 80-bit key register

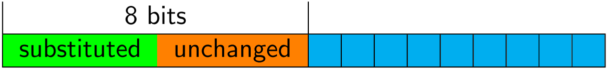
Key Update

1. Rotate 80-bit key register 61 bits to the left
   - Rotate 19 bits to the right instead
   - Use 2 *mov* instructions to rotate $2 \cdot 8 = 16$ bits
   - Use *ror* only for the last 3 bits
2. S-Box the top 4 bits of 80-bit key register
   - use a **byte** lookup table



Key Update

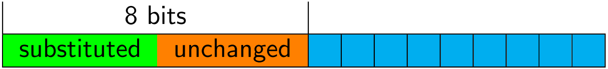
1. Rotate 80-bit key register 61 bits to the left
   - Rotate 19 bits to the right instead
   - Use 2 *mov* instructions to rotate $2 \cdot 8 = 16$ bits
   - Use *ror* only for the last 3 bits
2. S-Box the top 4 bits of 80-bit key register
   - use a **byte** lookup table



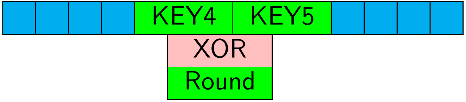
3. XOR key bits with round counter

Key Update

1. Rotate 80-bit key register 61 bits to the left
   - Rotate 19 bits to the right instead
   - Use 2 *mov* instructions to rotate $2 \cdot 8 = 16$ bits
   - Use *ror* only for the last 3 bits
2. S-Box the top 4 bits of 80-bit key register
   - use a **byte** lookup table

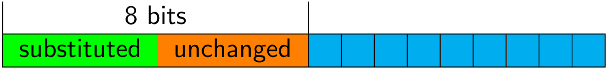


3. XOR key bits with round counter
   - XOR needs to span 2 registers

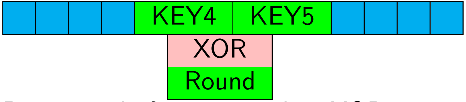


Key Update

1. Rotate 80-bit key register 61 bits to the left
   - Rotate 19 bits to the right instead
   - Use 2 *mov* instructions to rotate $2 \cdot 8 = 16$ bits
   - Use *ror* only for the last 3 bits
2. S-Box the top 4 bits of 80-bit key register
   - use a **byte** lookup table



3. XOR key bits with round counter
   - XOR needs to span 2 registers



   - Do step 3 before step 1 then XOR spans only 1 register

Serialization of the Algorithm

```
; state ^= roundkey
addRoundKey:
    eor STATE0, KEY0
    eor STATE1, KEY1
    eor STATE2, KEY2
    eor STATE3, KEY3
    eor STATE4, KEY4
    eor STATE5, KEY5
    eor STATE6, KEY6
    eor STATE7, KEY7
    ret
```

Serialization of the Algorithm

```
; half state ^= roundkey
addRoundKey:
    eor STATE0, KEY0
    eor STATE1, KEY1
    eor STATE2, KEY2
    eor STATE3, KEY3
    ret
```

This helps with:

- doing I/O
- applying round keys
- applying S-Boxes
- applying P-Layer

Serialization of the Algorithm

```
; half state ^= roundkey
addRoundKey:
    eor STATE0, KEY0
    eor STATE1, KEY1
    eor STATE2, KEY2
    eor STATE3, KEY3
    ret
```

This helps with:

- doing I/O
- applying round keys
- applying S-Boxes
- applying P-Layer

But we need I/O:

```
consecutive_input:
    ld STATE0, X+
    ld STATE1, X+
    ld STATE2, X+
    ld STATE3, X+
    ret

interleaved_output:
    st STATE3, X-
    dec X
    st STATE2, X-
    dec X
    st STATE1, X-
    dec X
    st STATE0, X-
    dec X
    ret
```

Indirect Register Addressing

```
; state ^= roundkey (full state in SRAM)
addRoundKey:
      clr YL                ; point Y at first key register
addRoundKey_byte:
      ld INPUT, X           ; load input
      ld KEY_BYTE, Y+       ; load key, advance pointer
      eor INPUT, KEY_BYTE   ; XOR
      st X+, INPUT          ; store output, advance pointer

      cpi YL, 8             ; loop over 8 bytes
      brne addRoundKey_byte

      subi XL, 8            ; point at the start of the block
      ret
```

Packed S-Boxes

Before:

C	5	6	B	9	0	A	D	3	E	F	8	4	7	1	2

Packed S-Boxes

Before:

C	5	6	B	9	0	A	D	3	E	F	8	4	7	1	2

After:

C5	6B	90	AD	3E	F8	47	12

Packed S-Boxes

Before:

C	5	6	B	9	0	A	D	3	E	F	8	4	7	1	2

After:

C5	6B	90	AD	3E	F8	47	12

```
unpack_sBox:
    asr ZL                  ; halve input, take carry
    lpm SBOX_OUTPUT, Z      ; get s-box output
    brcs odd_unpack         ; branch depending on carry
even_unpack:
    swap SBOX_OUTPUT        ; swap nibbles in s-box output
odd_unpack:
    cbr SBOX_OUTPUT, 0xF0 ; clear high nibble in s-box output
    ret
```

Packed S-Boxes

Before:

C	5	6	B	9	0	A	D	3	E	F	8	4	7	1	2

After:

C5	6B	90	AD	3E	F8	47	12

```
unpack_sBox:
    asr ZL
    lpm SBOX_OUTPUT, Z
    brcs odd_unpack          ; 2 cycles if true
even_unpack:
    swap SBOX_OUTPUT         ; 1 cycle
    rjmp unpack              ; 2 cycles
odd_unpack:
    nop                      ; 1 cycle
    nop
; 4 cycles total
unpack:
    cbr SBOX_OUTPUT, 0xF0
    ret
```

S-Box Optimization

```
sBoxByte:
    ; input (low nibble)
    mov ZL, INPUT           ; load s-box input
    cbr ZL, 0xF0            ; clear high nibble in input
    rcall unpack_sBox       ; get output in SBOX_OUTPUT
    cbr INPUT, 0xF          ; clear low nibble in output
    or INPUT, SBOX_OUTPUT   ; save low nibble to output
    ; fall through
sBoxHighNibble:
    mov ZL, INPUT           ; load s-box input
    cbr ZL, 0xF             ; clear low nibble in input
    swap ZL                 ; move high nibble to low nibble
    rcall unpack_sBox       ; get output in SBOX_OUTPUT
    swap SBOX_OUTPUT        ; move low nibble to high nibble
    cbr INPUT, 0xF0         ; clear high nibble in output
    or INPUT, SBOX_OUTPUT   ; save high nibble to output
    ret
```

S-Box Optimization

```
sBoxByte:
    rcall sBoxLowNibbleAndSwap ; apply s-box to low nibble
                               ; and swap nibbles
    rjmp sBoxLowNibbleAndSwap  ; do it again and return
sBoxHighNibble:
    swap INPUT                 ; swap nibbles in IO register
sBoxLowNibbleAndSwap:
    mov ZL, INPUT              ; load s-box input
    cbr ZL, 0xF0               ; clear high nibble in s-box input
    rcall unpack_sBox
    cbr INPUT, 0xF             ; clear low nibble in IO register
    or INPUT, SBOX_OUTPUT      ; save low nibble to IO register
    swap INPUT                 ; swap nibbles
    ret
```

S-Box Optimization

```
sBoxByte:
    rcall sBoxLowNibbleAndSwap ; apply s-box to low nibble
                               ; and swap nibbles
    rjmp sBoxLowNibbleAndSwap  ; do it again and return
sBoxHighNibble:
    swap INPUT                 ; swap nibbles in IO register
sBoxLowNibbleAndSwap:
    mov ZL, INPUT              ; load s-box input
    cbr ZL, 0xF0               ; clear high nibble in s-box input
    asr ZL                     ; halve input, take carry
    lpm SBOX_OUTPUT, Z         ; get s-box output
    brcs odd_unpack            ; branch depending on carry
even_unpack:
    swap SBOX_OUTPUT           ; swap nibbles in s-box output
odd_unpack:
    cbr SBOX_OUTPUT, 0xF0      ; clear high nibble in s-box output
    cbr INPUT, 0xF             ; clear low nibble in IO register
    or INPUT, SBOX_OUTPUT      ; save low nibble to IO register
    swap INPUT                 ; swap nibbles
    ret
```

P-Layer Nibble

```
pLayerNibble:
  ror INPUT   ; move bit into carry
  ror OUTPUT0 ; move bit into output register
  ror INPUT   ; etc
  ror OUTPUT1
  ror INPUT
  ror OUTPUT2
  ror INPUT
  ror OUTPUT3
  ret
```

- Apply twice to consume an input byte
- After 4 input bytes, 4 output bytes (half block) are filled
- Interleave 2 half blocks

2 Step P-Layer

Half state input

Half state output, interleaved

Second half state input

Second half state output, interleaved

Using SREG Flags and Stack

```
setup_redo_block:
        clt                  ; clear T flag
        rjmp redo_block      ; do the second part
block:
        set                  ; set T flag
        ; fall through
redo_block:
        ; instructions here happen twice when called from block

        brts setup_redo_block ; redo this block? (if T flag set)
        ret
```

1. Input
   - pLayerNibble and push 4 output bytes to stack
   - Do other half
2. Output
   - Point at last odd state byte
   - Pop from stack and save 4 output bytes
   - Point at last even state byte and do other half

Key Register Rotation

```
rotate_left_i:
  lsl KEY9                ; take MSB as carry, clear LSB
  rol KEY8                ; rotate MSB out, carry bit in
  rol KEY7                ; etc
  rol KEY6
  rol KEY5
  rol KEY4
  rol KEY3
  rol KEY2
  rol KEY1
  rol KEY0
  adc KEY9, ZERO          ; add carry bit to last key byte
  dec ITEMP               ; decrement counter
  brne rotate_left_i      ; loop
  ret
```

Key Register Rotation

```
rotate_left_i:
  ldi YL, 10            ; point at last key byte
  clc                   ; clear carry bit
rotate_left_i_bit:
  ld ROTATED_BITS, -Y   ; load key byte
  rol ROTATED_BITS      ; rotate bits
  st Y, ROTATED_BITS    ; save key byte
  cpse YL, ZERO         ; compare, skip if equal
  rjmp rotate_left_i_bit ; loop over all key bytes
  adc KEY9, ZERO        ; add carry bit to last key byte
  dec ITEMP             ; decrement counter
  brne rotate_left_i    ; loop
  ret
```

Numbers

	Encryption	Decryption	Size
AVR Crypto-lib	13225	18953	1514
Eisenbarth	1341	1405	936

Numbers

	Encryption	Decryption	Size
AVR Crypto-lib	13225	18953	1514
Eisenbarth	1341	1405	936
Speed-optimized	1091	-	1794
Size-optimized	23756	31673	272

Numbers

	Encryption	Decryption	Size
AVR Crypto-lib	13225	18953	1514
Eisenbarth	1341	1405	936
Speed-optimized	1091	-	1794
Bit sliced	370	-	4096
Size-optimized	23756	31673	272

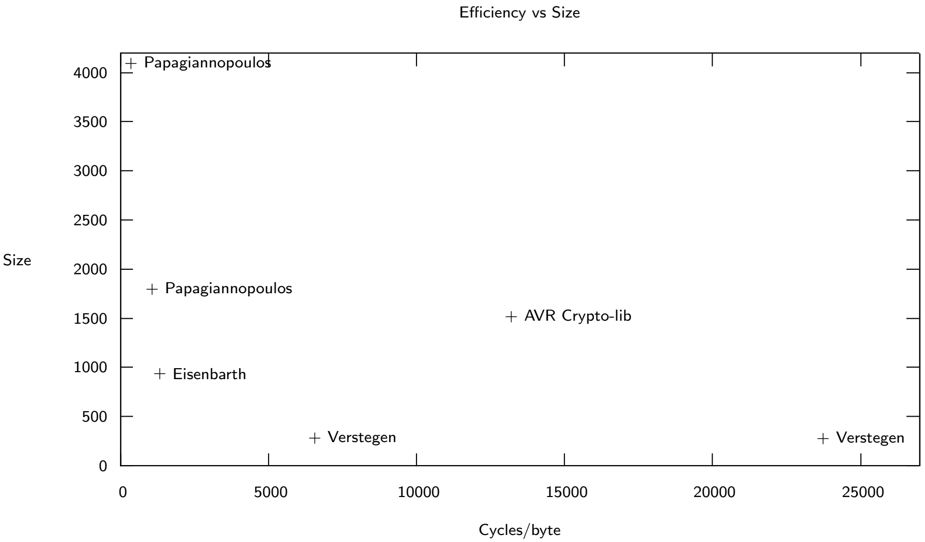
Numbers

	Encryption	Decryption	Size
AVR Crypto-lib	13225	18953	1514
Eisenbarth	1341	1405	936
Speed-optimized	1091	-	1794
Bit sliced	370	-	4096
Size-optimized	23756	31673	272
Unpacked S-Boxes	23361	31254	274
Inlined rotation	6973	9663	278
Inlined rotation, unpacked S-Boxes	6578	6578	280

Numbers

	Encryption	Decryption	Size
AVR Crypto-lib	13225	18953	1514
Eisenbarth	1341	1405	936
Speed-optimized	1091	-	1794
Bit sliced	370	-	4096
Size-optimized	23756	31673	272
Unpacked S-Boxes	23361	31254	274
Inlined rotation	6973	9663	278
Inlined rotation, unpacked S-Boxes	6578	6578	280
128-bit	35193	71467	272
Unpacked S-Boxes (128-bit)	34774	71002	274
Inlined rotation (128-bit)	8482	15419	290
Inlined rotation, unpacked S-Boxes (128-bit)	8064	14954	292

Relative Performance/Size



ASCII Art

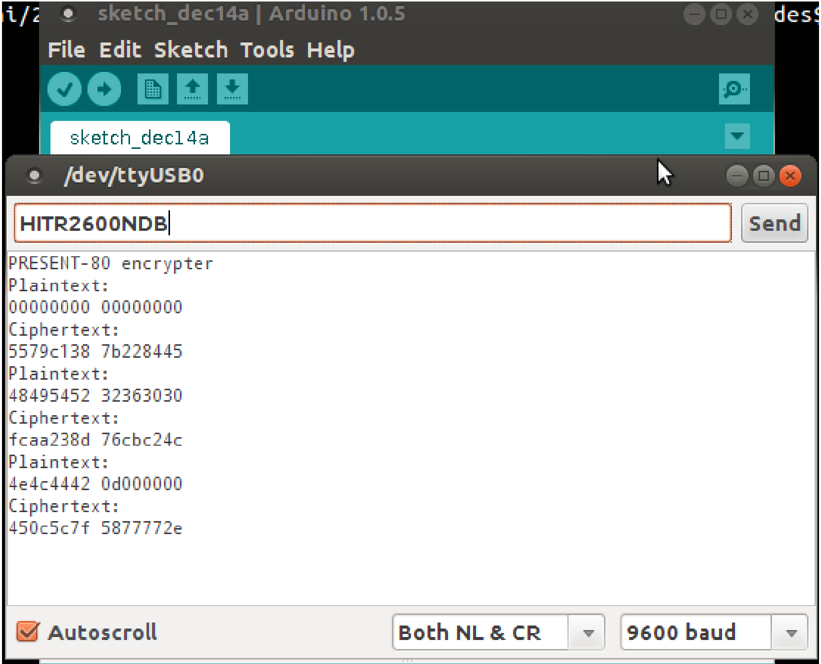
```
C56B90AD   3EF84712   5EF8C12    DB4630  79A57D0  3AD0    F1F  7F0E070E1
41D05DD05  CD047D080  2D16D00   82E81E1  06D0542  682E0   03D  04A9591F7
33C0CAE08  894CA9598  81991F9   883CD13  FACF9D1  E8A95   A9F  7089504D0
829   502  D08   295  089       5E8      2FE      F70E70  FE5     955
491  10F0  529   502  C00       0000     000      5F7080  7F8     52B
089587950  795879517  9587952   795879   5379508  9543958 6E0     D5D
F442687E3  D2DF802DD  DDF082E   4F31089  5CC278C  916 991 862     78D
93C830D1   F7A85008   9568E08     C91CD  DF8D936  A95 D9F7A85     008
954        427 F0E0   70E          0189  6DD      27C  C278D9     189
93C        A30 E1F7A  251      08   956  894      189  664E08     E91
CAD        FC9  DF6A  95D9F73  F932F931  F930F93  16F   4E894     F3C
F68        941   7966 4E08F91  8E93AA95  6A95D9F  71E   F4E89     419
96F        6CF   0895D7DFC5DF  CDDFE0D   FB7DFD9  F7C    0CF0     000
```

ASCII Art

```
s-boxes                                     decrypt (start+16)
|                                           |
C56B90AD   3EF84712   5EF8C12    DB4630  79A57D0  3AD0    F1F  7F0E070E1
41D05DD05  CD047D080  2D16D00   82E81E1  06D0542  682E0   03D  04A9591F7
33C0CAE08  894CA9598  81991F9   883CD13  FACF9D1  E8A95   A9F  7089504D0
829   502  D08   295  089       5E8      2FE      F70E70  FE5      955
491  10F0  529   502  C00       0000     000      5F7080  7F8      52B
089587950  795879517  9587952   795879   5379508  9543958 6E0      D5D
F442687E3  D2DF802DD  DDF082E   4F31089  5CC278C  916 991 862      78D
93C830D1   F7A85008   9568E08     C91CD  DF8D936  A95 D9F7A85      008
954        427 F0E0   70E          0189  6DD      27C  C278D9      189
93C        A30 E1F7A  251      08   956  894      189  664E08      E91
CAD        FC9  DF6A  95D9F73  F932F931  F930F93  16F   4E894      F3C
F68        941   7966 4E08F91  8E93AA95  6A95D9F  71E   F4E89      419
96F        6CF   0895D7DFC5DF  CDDFE0D   FB7DFD9  F7C    0CF0      000
                     |
                     encrypt (end-16)
```

S-Boxes, decrypt, rotate_left_i, sBoxByte, sBoxNibble, pLayerNibble, schedule_key, addRoundKey, sBoxLayer, setup, pLayer, encrypt.

Arduino sketch



Questions?

https://github.com/aczid/ru_crypto_engineering/

https://github.com/kostaspap88/PRESENT_speed_implementation/

https://github.com/aczid/PRESENT_arduino

https://github.com/kostaspap88/bitslicedpresent/