CHES 2014

FPGA Implementations of SPRING And Their Countermeasures against Side-Channel Attacks

Hai Brenner[1], Lubos Gaspar[2], Gaëtan Leurent[3], Alon Rosen[1], François-Xavier Standaert[2]

[1] Interdisciplinary Center, Herzliya, Israel
[2] Crypto group, Université catholique de Louvain, Louvain-la-Neuve, Belgium
[3] Inria, EPI SECRET, Rocquencourt, France

haibrenner@gmail.com, lubos.gaspar@uclouvain.be, gaetan.leurent@inria.fr, alon.rosen@idc.ac.il, fstandae@uclouvain.be

Road Map

- Introduction to SPRING PRF
- Unprotected hardware implementation
- Countermeasures against side-channel attacks
  - Fully masked solution
  - Hybrid solution

Introduction to SPRING PRF

Function Description

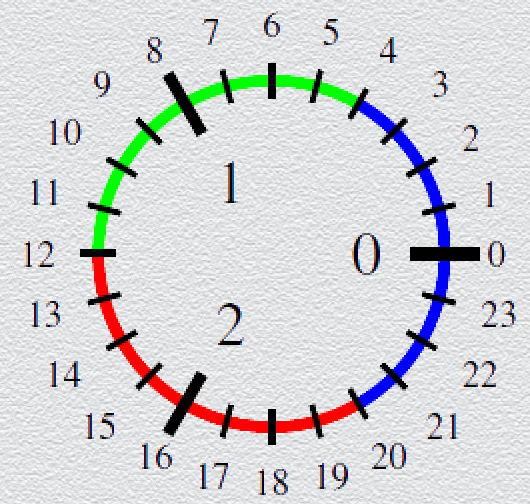
- SPRING is **S**ubset-**P**roduct with **R**ounding over a **RING**

$$F_{a,s}(x_1, \dots, x_k) = B\left(a \cdot \prod_{i=1}^{k} s_i^{x_i}\right)$$

- a, s_{i} - a vector of polynomials in polynomial ring $R = \mathbb{Z}_q[x]/(x^n + 1)$ (all coefficients in $[0, q-1]$).
- Input - k bits.
- B- a rounding function, coefficient-wise.

Rounding [BPR'12]

- Idea:
  Generate errors deterministically by rounding $\mathbb{Z}_q$ to "sparse" subset (e.g. $\mathbb{Z}_p$).
  Let $p < q$ and define $\lfloor x \rceil_p = \lfloor (p/q) \cdot x \rceil \; mod \; p$



- Interpretation: rounding discards low-order bits of coefficient.

- Claim: We infer hardness of SPRING by reduction from LWE (Learning With Errors) – NP hard problem.

Concrete Function Parameters

$$F_{a,s}(x_1,\ldots,x_{64}) = BCH\left(\left\lfloor a \cdot \prod_{i=1}^{64} s_i^{x_i} \right\rceil_2\right)$$

- Highly optimized parameters:
  - $q = 257$
  - Polynomial degree $n = 128$ (security parameter)
  - $k = |x| = 64$
  - B– rounding to $\mathbb{Z}_2$ (whether coefficient smaller than $q/2$ or not)
  - Dual-BCH (ECC) w/ parameters $[128{,}64{,}22]$ reduces output bias.
  - CTR mode (amortized computation of consecutive subset-products).

Optimizations for SPRING

- Optimizing subset product: $a \cdot \prod_{i=1}^{64} s_i^{x_i}$
  - FFT Instead of regular multiplication

    $$a \cdot s_1^{x_1} \cdot \ldots \cdot s_{64}^{x_{64}} = F^{-1}\left(F(a) \otimes F(s_1)^{x_1} \otimes \ldots \otimes F(s_{64})^{x_{64}}\right)$$

    $\otimes$ is point-wise multiplication.
  - Pre-compute F(a), F(s_i)
  - Replace F(a), F(s_i) entries w/ discrete logs → multiplications replaced by point-wise additions.
  - Subset sum modulo $q - 1 = 256$. Simply ignore carry of the most significant bit → modulo operation "for free".
  - Convert back discrete logs to polynomial coefficients w/ 256 entries LUT.

Optimizations for SPRING (continued)

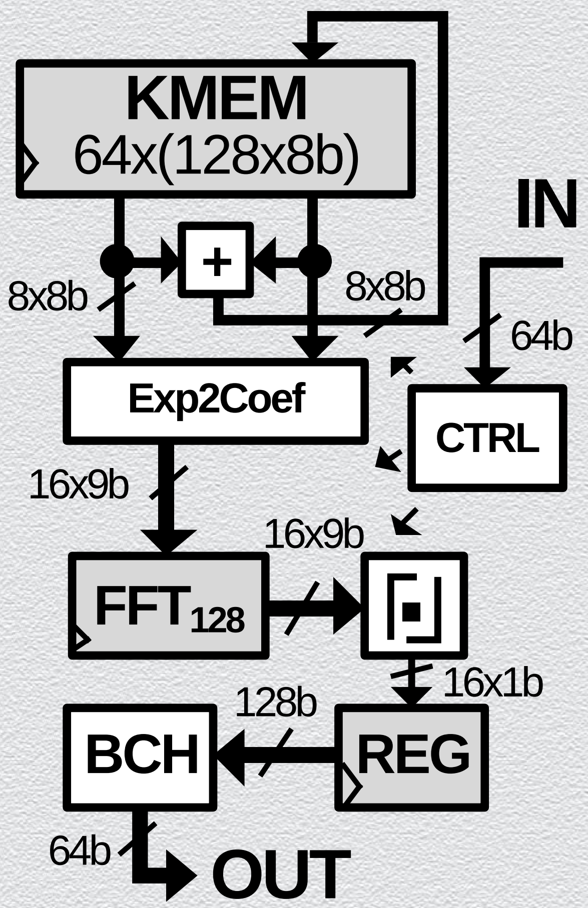
- Some other optimizations:
  - Dual-BCH code:
    - Simple & efficient ECC.
    - Just compute syndrome of result w/ dual of generating polynomial of code.
    - simply just 29 shifts and xors.
  - CTR mode: (Gray-Code)
    - Amortized computation of subset-sum.
    - Update subset-sum with only a single add/subtract key element to previous computation each round.

```
Gray-Code
counter
      0000
      0001
      0011
      0010
      0110
      0111
      0101
      0100
      1100
      1101
      1111
      1110
      1010
      1011
      1001
      1000
```

Unprotected hardware implementation

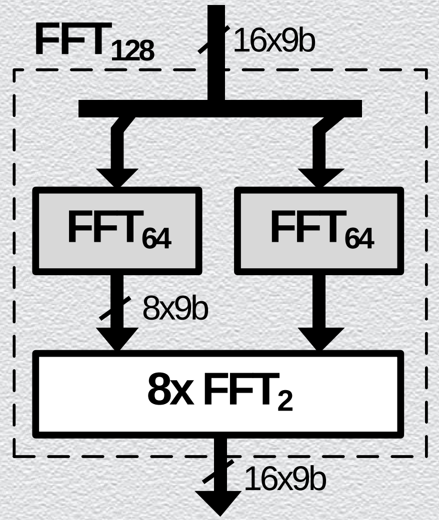
Top level architecture (for Xilinx Virtex 6 FPGA)

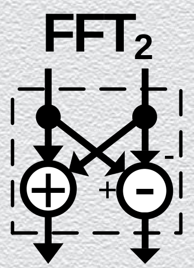
- Arithmetic op. mod $q = 257$ and exponent arithmetic op. mod $q - 1 = 256$
- Exponents associated with FFT coefficients stored in True Dual Port RAM (KMEM)
- Subset-sum computed on 8 entries **in parallel**
- Exponents transformed to coeffs. using LUTs
- FFT_{128} + Rounding performed in 36 clock cycles
- Next Subset-sum computed in parallel with FFT_{128}

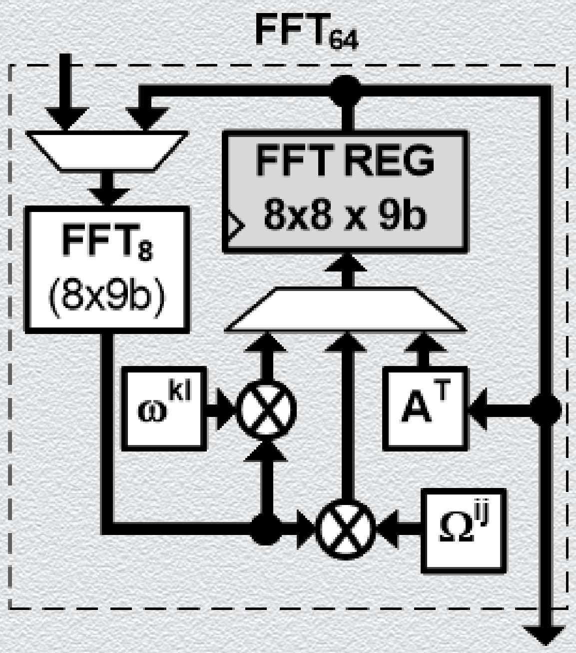


Construction of FFT_{128}

- FFT_{128} processes 16 coefficients in parallel
- FFT_{128} composed of: 2x FFT_{64} sequential units, 8x FFT_2 combinatorial units
- FFT_{64} is implemented using FFT_8 and a register
- FFT_{64} processes 8x8 matrix of coefficients
- FFT_{64} units also include matrix transpose and constant multiplications

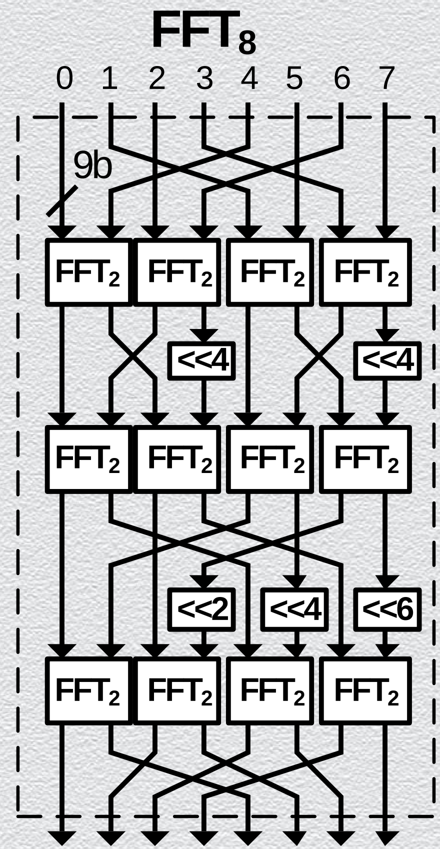


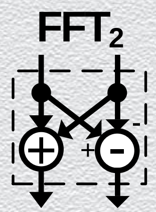




Decomposition of FFT_8

- Combinatorial
- Processes 8 coefficients in parallel
- 3 layers of FFT_2 and multiplications by constant powers of root of unity. $\omega = 139.\ \ \omega^{16} = 4$
- Constant multiplications implemented as bit rotations around 9-bit word
- FFT_2 contains: 9-bit combinatorial adder and substractor mod $q\ =\ 257$





Cost evaluation & timing results

- Synthesized for Xilinx Virtex 6 FPGA
- Size
  - FFT unit occupies 76% of SPRING area
  - Constant multipliers inside FFT are the most expensive (69% of FFT)
  - SPRING occupies only 4% of FPGA resources
- Speed
  - Evaluation in only 40 clock cycles

Units	Slices	BRAM (36kb)
KMEM	0	2
Subset sum	16	0
Exp2Coef	128	0
FFT_{128} total	1258	0
$\rightarrow$ 2x FFT_8	210	0
$\rightarrow$ 2x FFT REG + transpose	110	0
$\rightarrow$ 2x Mult. $\Omega^{i,j}$	496	0
$\rightarrow$ 1x Mult. $\omega^{k,l}$	378	0
$\rightarrow$ 8x FFT_2	64	0
Rounding + REG	32	0
BCH	189	0
Control logic	27	0
SPRING - TOTAL	**1650**	**2**

Cost and performance comparison

- 33x faster than Lapin
- 10x faster than Spring on Intel Core i7

Algorithm	Type	Datapath	LUT	FF	BRAM	DSP	F_{max}^{a}	Cycles
SPRING	PRF	128/144b	7292	294	2x36k	0	91.7	40
Lapin[1]	Auth.	128b	742	140	6x36k	0	140.3	1332
Comp-LWE[2]	PKE	N/A	1879	1142	3x18k	1	250.0	13287[b]
AES-LUT[3]	PRP	128b	933	399	10x18k	0	674.0	11
AES-COMB[3]	PRP	128b	2335	535	0	0	218.6	11
AES-COMB[3]	PRP	32b	467	976	0	0	315.1	58
SPRING[4]	PRF	64b	Software implementation					392

[1] L. Gaspar, G. Leurent, FX. Standaert: Hardware Implementation and Side-Channel Analysis of Lapin, CT-RSA'14

[2] S.S Roy, F. Vercauteren, N. Mentens, D.D. Chen, I. Verbauwhede: Compact Ring-LWE based Cryptoprocessor, ePrint 2013/866

[3] Crypto group, UCL, Louvain-la-Neuve, Belgium

[4] Software implementation on Intel Core i7 Ivy Bridge

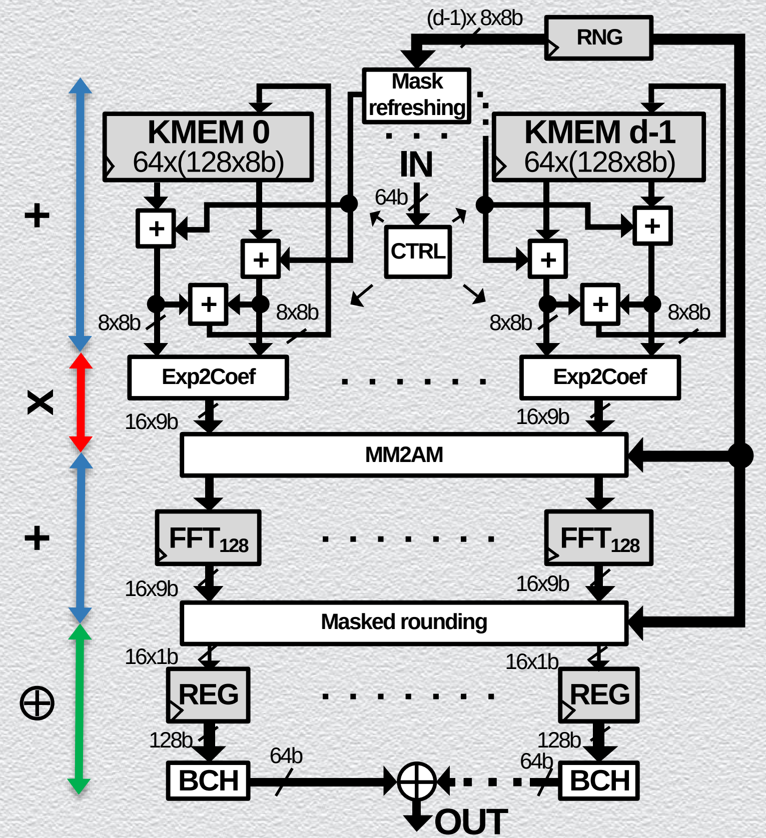
[a] Maximum frequency is denoted in MHz

[b] Number of clock cycles for encryption only

Countermeasures against side-channel attacks

Fully masked SPRING

- **Key exponents** - masked by additive shares
- **Subset sum** (linear)
- **Exp2Coef** - additive masking is changed to multiplicative masking
- **MM2AM** – Sync step. Exchange between shares. Regain additive shares
- **FFT** (linear)
- **Masked rounding** - rounding is highly non-linear. Complex sync with exchanges. Generates Boolean shares.
- **BCH** (linear)



Note:

+ Additive sharing
x Multiplicative sharing
⊕ Boolean sharing

Fully masked SPRING - cost

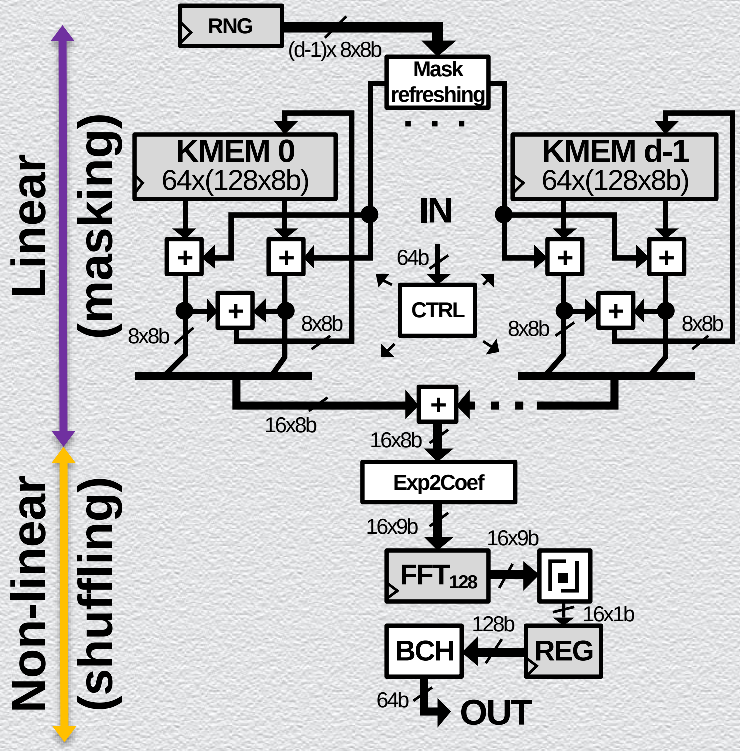
	Basic operations					Random	Total # of slices			
	ADD	MUL	INV	MUX2	XOR	bits	d=2	d=3	d=4	d=5
Msk. Refresh	d-2	0	0	0	0	8(d-1)	3	5	6	7
MM2AM	d-2	$3d^2$-2d	d	0	0	$8(d^2-1)$	527	1353	2551	4121
Msk. round	3d-2	0	0	256d	d-1	266d-257	1321	1409	1473	1894

- Coordination between parties:
  - Expensive resources
  - Slow: <u>clock cycle increase quadratically with number of shares</u>
  - Requires lot of fresh randomness

- Fits into FPGA, but not practical

Partially masked SPRING - idea

- After subset sum computation, <u>diffusion</u> of secret key is complete
- Fast: clock cycles increase linearly with number of shares
- Next, it is sufficient to protect other units only against Simple Power Analysis (SPA)
- <u>Shuffling</u> the state is efficient countermeasure:
  - Shuffling 8 rows provide 8! = 40320 execution permutations → sufficient against SPA
  - Shuffling implementation size is negligible (only adds 24 slices)
  - Has no impact on performance



Conclusions

Conclusions

- **The SPRING PRF**
  - Simple algebraic structure
  - Highly parallelizable and easy to mask
- **Unprotected SPRING**
  - First SPRING hardware implementation
  - Compact and very fast
- **Fully masked implementation**
  - Complexity of frequent masking is a limiting factor
  - Used area and random bits increase <u>quadratically</u> w/ # of shares.
- **Partially masked implementation**
  - Only subset sum is masked (necessary protection against DPA)
  - The rest is shuffled (sufficient protection against SPA)

Thank you for attention!