SPRING

Fast Pseudorandom Functions from Rounded Ring Products

Abhishek Banerjee[1] Hai Brenner[2] Gaëtan Leurent[3]
Chris Peikert[1] Alon Rosen[2]

[1] Georgia Institute of Technology

[2] IDC Herzliya

[3] UCL & Inria

FSE 2014

Motivation

Public key

- Strong algebraic structure
- Security reduction
- Slow

Secret key

- Security from cryptanalysis
- Fast

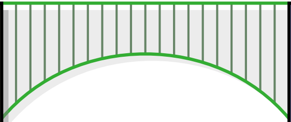
Bridging the gap

- Can we have an efficient design with strong algebraic structure?
  - Security reduction from a well-understood problem?
  - Extra features?
  - Previous examples: SWIFFT, FSB, Lapin, HB family

Motivation

Public key

- Strong algebraic structure
- Security reduction
- Slow



Secret key

- Security from cryptanalysis
- Fast

Bridging the gap

- Can we have an efficient design with strong algebraic structure?
    - Security reduction from a well-understood problem?
    - Extra features?
    - Previous examples: SWIFFT, FSB, Lapin, HB family

SPRING construction

Subset Product with Rounding over a ring

$$F_{a,\vec{s}}(x_1, \dots, x_k) := S\left(a \cdot \prod_{j=1}^{k} s_j^{x_j}\right)$$

- Lattice-based PRF [BPR, Eurocrypt '12]
- Polynomial ring $R_p = \mathbb{Z}_p[X]/(X^n + 1)$
- Key: $a, (s_i)_{i=1}^{k} \in R_p$
- Rounding function S
  - *e.g.* MSB of each polynomial coefficient

SPRING security

- Based on the RING-LEARNING WITH ERRORS assumption
  - Secret polynomial $s \in R_p$, $\quad R_p = \mathbb{Z}_p[X]/(X^n + 1)$
  - Distinguish $(a_i, a_i \cdot s + e_i)$ from uniform
  - *Reduction to worst-case ideal lattice problems*
- Deterministic version: RING-LEARNING WITH ROUNDING assumption
  - Secret polynomial $s \in R_p$
  - Distinguish $(a_i, \lfloor a_i \cdot s \rceil)$ from uniform
  - Rounding removes information, like adding noise
- Two SPRING outputs gives something similar to an LWR sample
  - $F_{a,\vec{s}}(x_1, \ldots, x_k) := S\left(a \cdot \prod_{j=1}^{k} s_j^{x_j}\right)$
  - Secret polynomials s, t
  - Output $(\lfloor t \rceil, \lfloor t \cdot s \rceil)$

SPRING security

- Based on the Ring-Learning With Errors assumption
  - Secret polynomial $s \in R_p$, $R_p = \mathbb{Z}_p[X]/(X^n + 1)$
  - Distinguish $(a_i, a_i \cdot s + e_i)$ from uniform
  - *Reduction to worst-case ideal lattice problems*

- Deterministic version: Ring-Learning With Rounding assumption
  - Secret polynomial $s \in R_p$
  - Distinguish $(a_i, \lfloor a_i \cdot s \rceil)$ from uniform
  - Rounding removes information, like adding noise

- Two SPRING outputs gives something similar to an LWR sample
  - $F_{a,\vec{s}}(x_1, \dots, x_k) := S\left(a \cdot \prod_{j=1}^{k} s_j^{x_j}\right)$
  - Secret polynomials s, t
  - Output $(\lfloor t \rceil, \lfloor t \cdot s \rceil)$

From provable security to efficiency

- Security reduction require huge parameters
- What happens when we use small parameters?
  - Security reduction not applicable as such
  - Guideline towards reasonable constructions (mode of operation?)
  - Bias can appear (was negligible with large parameters)
  - Concrete security evaluation needed

Choice of ring

SPRING

$$F_{a,\vec{s}}(x_1, \dots, x_k) := S\left(a \cdot \prod_{j=1}^{k} s_j^{x_j}\right) \quad \text{over } R_p = \mathbb{Z}_p[X]/(X^n + 1)$$

- Select parameters with fast polynomial product
  1. Polynomial product very efficient using FFT algorithm
  2. Arithmetic mod $2^i + 1$ is efficient in software
- Problem was studied for SWIFFT
  - Use $p = 257$, $n = 128$

Product in the ring R_{257}

Fast polynomial product $h = f \cdot g$

1. Evaluate f and g: $f_i = f(x_i)$, $g_i = g(x_i)$ (256 points)
2. Multiply values coefficients-wise
3. Interpolate h s.t. $h(x_i) = f_i \times g_i$ (degree 256)

- Let ω be a 256-th root of unity, $x_i = \omega^i$, $\omega = 41$
  Use FFT for evaluation/interpolation in $n \log(n)$

- We want $f \cdot g \bmod x^{128} + 1$
  - $x^{128} + 1 = \prod(x - \omega^{2i+1})$
  - Chinese Remainder: compute $h \bmod x - \omega^{2i+1}$ *i.e.* $h(\omega^{2i+1})$
- Evaluating $f(\omega^{2i+1})$
  - $\phi : \sum b_i \cdot x^i \mapsto \sum (b_i \cdot \omega^i) \cdot x^i$
  - $\phi(f)(\omega^{2i}) = f(\omega^{2i+1})$
- $\mathrm{FFT}_{128}(\phi(f \cdot g)) = \mathrm{FFT}_{128}(\phi(f)) \times \mathrm{FFT}_{128}(\phi(g))$ (coeff.-wise $\times$)

Product in the ring R_{257}

Fast polynomial product $h = f \cdot g$

1. Evaluate f and g: $f_i = f(x_i)$, $g_i = g(x_i)$ (256 points)
2. Multiply values coefficients-wise
3. Interpolate h s.t. $h(x_i) = f_i \times g_i$ (degree 256)

- Let ω be a 256-th root of unity, $x_i = \omega^i$, $\omega = 41$
  Use FFT for evaluation/interpolation in $n \log(n)$

- We want $f \cdot g \bmod x^{128} + 1$
  - $x^{128} + 1 = \prod(x - \omega^{2i+1})$
  - Chinese Remainder: compute $h \bmod x - \omega^{2i+1}$ *i.e.* $h(\omega^{2i+1})$
- Evaluating $f(\omega^{2i+1})$
  - $\phi : \sum b_i \cdot x^i \mapsto \sum (b_i \cdot \omega^i) \cdot x^i$
  - $\phi(f)(\omega^{2i}) = f(\omega^{2i+1})$
- $FFT_{128}(\phi(f \cdot g)) = FFT_{128}(\phi(f)) \times FFT_{128}(\phi(g))$ (coeff.-wise $\times$)

Product in the ring R_{257}

Fast polynomial product $h = f \cdot g$

1. Evaluate f and g: $f_i = f(x_i)$, $g_i = g(x_i)$ (256 points)
2. Multiply values coefficients-wise
3. Interpolate h s.t. $h(x_i) = f_i \times g_i$ (degree 256)

- Let ω be a 256-th root of unity, $x_i = \omega^i$, $\omega = 41$
  Use FFT for evaluation/interpolation in $n \log(n)$

- We want $f \cdot g$ mod $x^{128} + 1$
  - $x^{128} + 1 = \prod(x - \omega^{2i+1})$
  - Chinese Remainder: compute h mod $x - \omega^{2i+1}$ *i.e.* $h(\omega^{2i+1})$
- Evaluating $f(\omega^{2i+1})$
  - $\phi : \sum b_i \cdot x^i \mapsto \sum (b_i \cdot \omega^i) \cdot x^i$
  - $\phi(f)(\omega^{2i}) = f(\omega^{2i+1})$
- $\mathrm{FFT}_{128}(\phi(f \cdot g)) = \mathrm{FFT}_{128}(\phi(f)) \times \mathrm{FFT}_{128}(\phi(g))$ (coeff.-wise $\times$)

Product in the ring R_{257}

Fast polynomial product $h = f \cdot g \bmod x^{128} + 1$

1. Evaluate f and g: $f_i = f(x_i), g_i = g(x_i)$ (128 points)
2. Multiply values coefficients-wise
3. Interpolate h s.t. $h(x_i) = f_i \times g_i$ (degree 128)

- Let ω be a 256-th root of unity, $x_i = \omega^{2i+1}$, $\omega = 41$
  Use FFT for evaluation/interpolation in $n \log(n)$

- We want $f \cdot g \bmod x^{128} + 1$
  - $x^{128} + 1 = \prod(x - \omega^{2i+1})$
  - Chinese Remainder: compute $h \bmod x - \omega^{2i+1}$ *i.e.* $h(\omega^{2i+1})$
- Evaluating $f(\omega^{2i+1})$
  - $\phi : \sum b_i \cdot x^i \mapsto \sum(b_i \cdot \omega^i) \cdot x^i$
  - $\phi(f)(\omega^{2i}) = f(\omega^{2i+1})$
- $\mathrm{FFT}_{128}(\phi(f \cdot g)) = \mathrm{FFT}_{128}(\phi(f)) \times \mathrm{FFT}_{128}(\phi(g))$ (coeff.-wise $\times$)

Implementation tricks

SPRING PRF

$$F_{a,\vec{s}}(x_1, \dots, x_k) := S\left(a \cdot \prod_{j=1}^{k} s_j^{x_j}\right)$$

- Use FFT for the subset product
  - $\prod_{x_j=1} s_j = \phi^{-1}\left(\mathrm{FFT}^{-1}\left(\bigtimes_{x_j=1} \mathrm{FFT}(\phi(s_j))\right)\right)$
  - Store $\tilde{s}_j := \mathrm{FFT}(\phi(s_j))$ (equivalent key)
  - $\prod_{x_j=1} s_j = \phi^{-1}\left(\mathrm{FFT}^{-1}\left(\bigtimes_{x_j=1} \tilde{s}_j\right)\right)$ (coefficients-wise product)
- Use counter mode for a stream cipher
  - Single addition instead of subset-sum

Implementation tricks

SPRING PRF

$$F_{a,\vec{s}}(x_1, \dots, x_k) := S\left(a \cdot \prod_{j=1}^{k} s_j^{x_j}\right)$$

- Use FFT for the subset product
  - $\prod_{x_j=1} s_j = \phi^{-1}\left(\mathrm{FFT}^{-1}\left(\bigtimes_{x_j=1} \mathrm{FFT}(\phi(s_j))\right)\right)$
  - Store $\widehat{s_{ij}} := \log\left(\widetilde{s_{ij}}\right), \tilde{s}_j := \mathrm{FFT}(\phi(s_j))$ (equivalent key)
  - $\prod_{x_j=1} s_j = \phi^{-1}\left(\mathrm{FFT}^{-1}\left(\exp\left(\sum_{x_j=1} \widehat{s_j}\right)\right)\right)$ (coefficients-wise product)
- Use counter mode for a stream cipher
  - Single addition instead of subset-sum

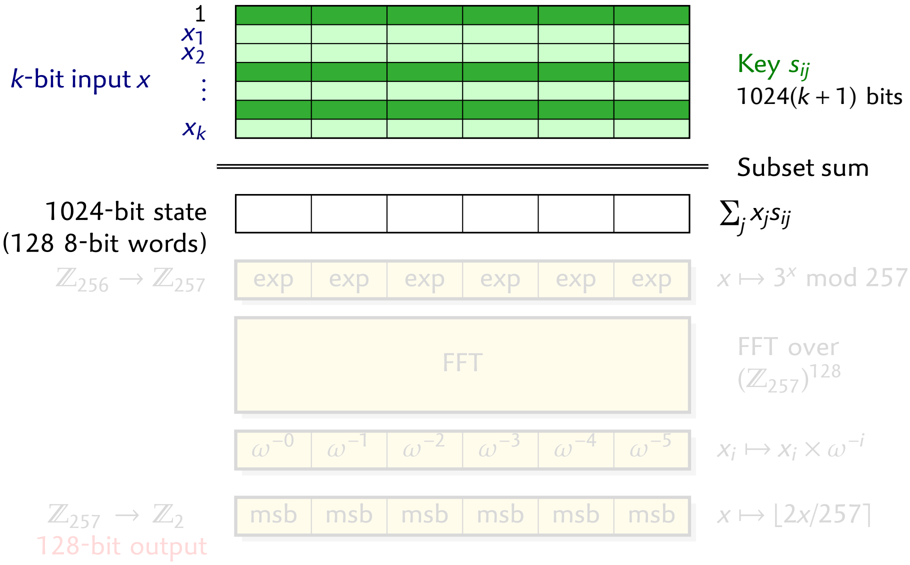
Implementation tricks

SPRING PRF

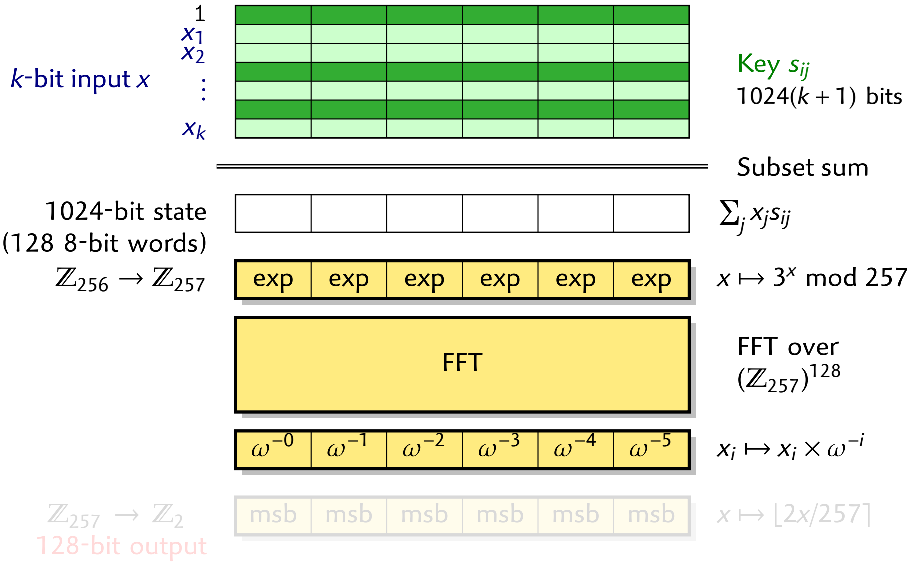
$$F_{a,\vec{s}}(x_1, \ldots, x_k) := S\left(a \cdot \prod_{j=1}^{k} s_j^{x_j}\right)$$

- Use FFT for the subset product
  - $\prod_{x_j=1} s_j = \phi^{-1}\left(\mathrm{FFT}^{-1}\left(\bigtimes_{x_j=1} \mathrm{FFT}(\phi(s_j))\right)\right)$
  - Store $\widehat{s_{ij}} := \log\left(\widetilde{s_{ij}}\right), \tilde{s}_j := \mathrm{FFT}(\phi(s_j))$ (equivalent key)
  - $\prod_{x_j=1} s_j = \phi^{-1}\left(\mathrm{FFT}^{-1}\left(\exp\left(\sum_{x_j=1} \widehat{s_j}\right)\right)\right)$ (coefficients-wise product)
- Use counter mode for a stream cipher
  - Single addition instead of subset-sum

SPRING over R_{257} ($p = 257, n = 128$)



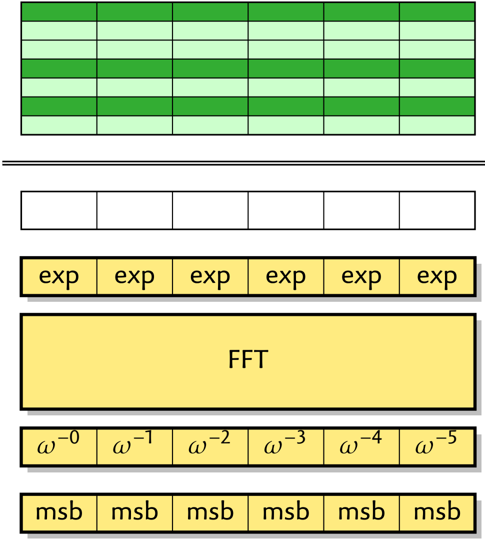
SPRING over R_{257} ($p = 257, n = 128$)



SPRING over R_{257} ($p = 257$, $n = 128$)

k-bit input x: $1, x_1, x_2, \vdots, x_k$

Key s_{ij}
$1024(k+1)$ bits



Subset sum

1024-bit state (128 8-bit words) — $\sum_j x_j s_{ij}$

$\mathbb{Z}_{256} \to \mathbb{Z}_{257}$ — $x \mapsto 3^x \bmod 257$

FFT over $(\mathbb{Z}_{257})^{128}$

$x_i \mapsto x_i \times \omega^{-i}$

$\mathbb{Z}_{257} \to \mathbb{Z}_2$ — $x \mapsto \lfloor 2x/257 \rceil$

128-bit output

Tweaks to the construction

Problems because of the small parameters

1. Polynomial are non-inversible with high probability
   - Product in a subspace
   - Use only units for the key elements
2. Rounding from $\mathbb{Z}_{257}$ has a bias 1/257
   - Output bits biased
   - Combine bits to reduce bias: SPRING-BCH
   - Or use $\mathbb{Z}_{514}$: SPRING-CRT

Tweaks to the construction

Problems because of the small parameters

1. Polynomial are non-inversible with high probability
   - Product in a subspace
   - Use only units for the key elements
2. Rounding from $\mathbb{Z}_{257}$ has a bias 1/257
   - Output bits biased
   - Combine bits to reduce bias: SPRING-BCH
   - Or use $\mathbb{Z}_{514}$: SPRING-CRT

Tweaks to the construction

Problems because of the small parameters

1. Polynomial are non-inversible with high probability
   - Product in a subspace
   - Use only units for the key elements
2. Rounding from $\mathbb{Z}_{257}$ has a bias 1/257
   - Output bits biased
   - Combine bits to reduce bias: SPRING-BCH
   - Or use $\mathbb{Z}_{514}$: SPRING-CRT

Tweaks to the construction

Problems because of the small parameters

1. Polynomial are non-inversible with high probability
   - Product in a subspace
   - Use only units for the key elements
2. Rounding from $\mathbb{Z}_{257}$ has a bias 1/257
   - Output bits biased
   - Combine bits to reduce bias: SPRING-BCH
   - Or use $\mathbb{Z}_{514}$: SPRING-CRT

SPRING-BCH

- Reduce the bias by combining output bits
  - Piling-up lemma: $\text{bias}(a \oplus b) = \text{bias}(a) \cdot \text{bias}(b)$
- Multiply with the transpose of the generating matrix of a code
  - Syndrome for the dual code
  - Any linear combination of output bits is the sum of d biased bits
  - Bias reduced exponentially in d
- We use an extended BCH code
  - Efficient
  - Best known distance
- Efficiency loss: only 64 output bits

SPRING-CRT

- Use the ring $R_{514} = \mathbb{Z}_{514}[X]/(X^n + 1)$
  - Unbiased rounding from $\mathbb{Z}_{514}$
- Chinese Remainder decomposition: $R_{514} \cong R_{257} \times R_2$
  - Compute modulo 257 and modulo 2, combine outputs
- Computation in R_2:
  - Efficient algorithms for subset-product in the paper
  - In counter mode: single multiplication using `PCLMUL`, or tables

Implementation

- Implementation using SIMD instructions
  - Compute operations in parallel on vector of data
  - SSE2 on Intel/AMD x86: desktop (Core) and embedded (Atom)
  - NEON on ARM: embedded CPU (Cortex A in smartphones, tablets)
- Subset sum optimized with precomputed tables
  - 2-bit inputs: $[0, s_0, s_1, s_0 + s_1]$
  - 8-bit inputs: 256 entries
- Multiplication in R_2 using `PCLMUL` instruction (if available), or precomputed tables
- Bottleneck is FFT

FFT implementation tricks

- Reuse efficient FFT from the SIMD hash function
- Decompose FFT as a two-dimensional FFT
  - Parallel FFT on lines and columns
- Elements in $\mathbb{Z}_{257}$ as 16-bit words
- Partial reduction mod257 with `(x&256) - (x>>8)`
  - Output in $[-127, 383]$
- Multiplication in $\mathbb{Z}_{257}$ using 16-bit signed multiplication
  - Reduce operands to $[-128, 128]$ beforehand

Performance

- 20-30 cycle/byte on Core i7 using SSE
  - Slow for a stream cipher, fast enough for practical use
- SPRING-CRT-CTR is about 4.5 times slower than AES-CTR
  - Excluding hardware AES instructions
  - Same ratio on a range of architectures

	SPRING-BCH		SPRING-CRT		AES-CTR	
	Single	CTR	Single	CTR	~~AES-NI~~	AES-NI
ARM Cortex A15	220	170	250	77	17.8	N/A
Atom	247	137	235	76	17	N/A
Core i7 Nehalem	74	60	76	29.5	6.9	N/A
Core i7 Ivy Bridge	60	46	62	23.5	5.4	1.3

Conclusion

SPRING: Subset Product with Rounding over a ring

- Strong algebraic structure
  - Simple design
    - Subset sum, table lookup, FFT, table lookup with small output
  - Large linear part good for masking, MPC
- Based on a design with security reduction
  - Security reduction does not apply with small parameters
  - Cryptanalysis is needed to evaluate the security
  - Expected security: about 128 bit
- High parallelism
  - Reasonable performances with vector instructions
  - Good performances in hardware?

Pseudo-code for SPRING

Implementation

Key: $(\widehat{a_i})_{i=0}^{127}, (\widehat{s_{ij}})_{i=0}^{127}{}_{j=0}^{k-1} \in \mathbb{Z}_{256}$

Input: $x_1, x_2, \dots x_k \in \{0,1\}$

for $0 \le i < k$ **do**
  $u_i \leftarrow \widehat{a_i} + \sum_j x_j \widehat{s_{ij}} \bmod 256$
  $u_i \leftarrow 3^{u_i} \bmod 257$
$\vec{u} \leftarrow \mathrm{FFT}_{128}^{-1}(\vec{u})$
for $0 \le i < k$ **do**
  $u_i \leftarrow u_i \cdot \omega^{-i} \bmod 257$
  $y_i \leftarrow \lfloor 2 \cdot u_i / 257 \rceil$
return $\vec{y}$