Higher-Order Masking in Practice: A Vector Implementation of Masked AES for ARM NEON

Junwei Wang, Praveen Kumar Vadnala,

Johann Großschädl, Qiuliang Xu

Shandong University, University of Luxemborg

CT-RSA 2015, April 20 - 24, 2015

Outline

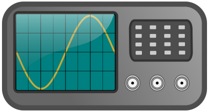
Outline

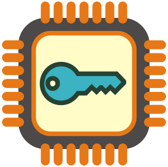
- Introduction
  - Differential Power Analysis
  - Masking Countermeasures
  - High-Order DPA Attacks
- Background
  - Advanced Encryption Standard
  - High-Order Masking
  - Rivain-Prouff Countermeasure
- Implementation
  - ARM NEON
  - Performance-Critical Analysis
  - Implementation of Secure Field Multiplication
- Results and Comparison
- Conclusion

Introduction

Workstation

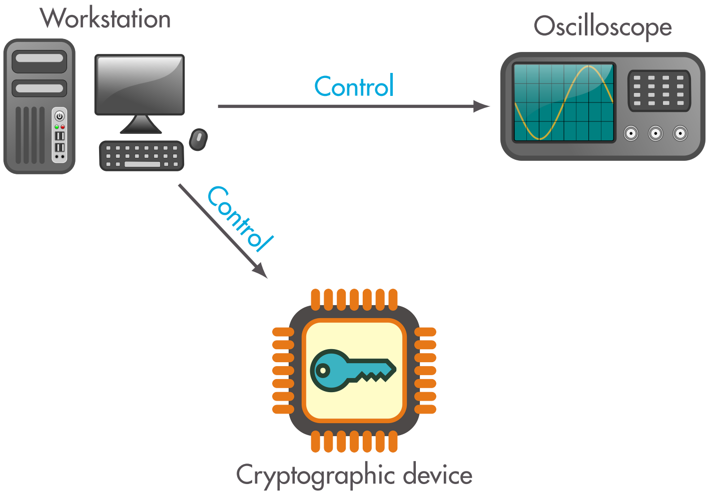
Oscilloscope



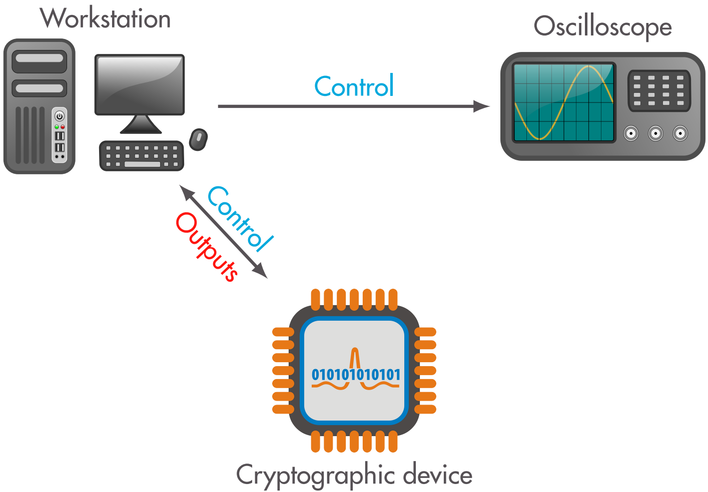


Cryptographic device

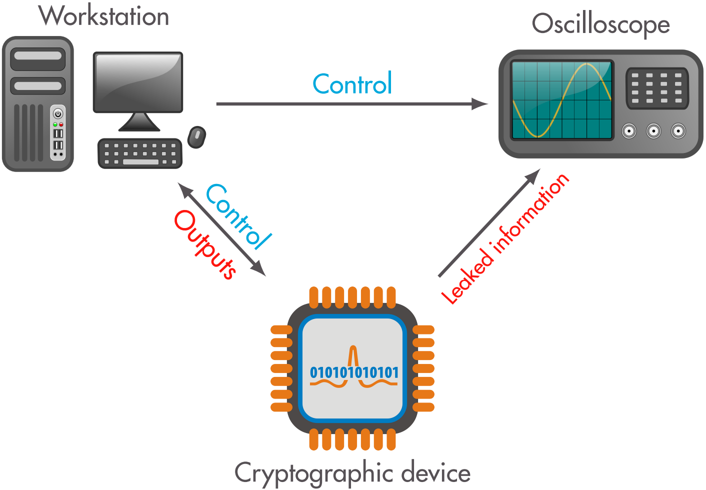
Introduction



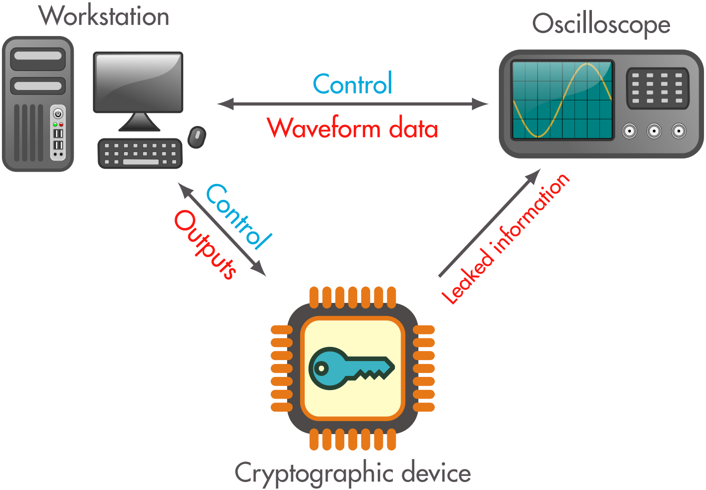
Introduction

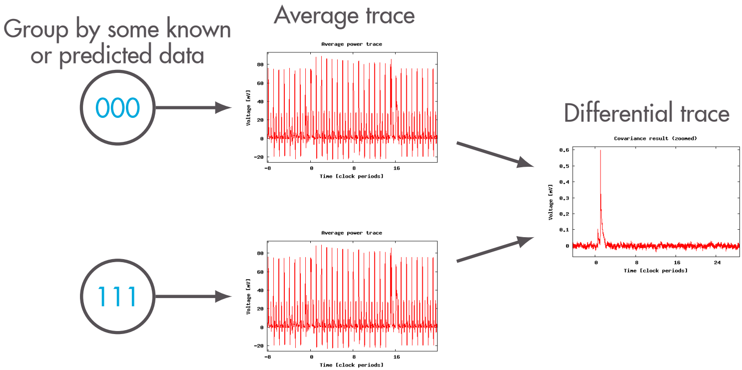


Introduction



Introduction





Group by some known or predicted data
000
111
Average trace
Average power trace
Voltage [mV]
Time [clock periods]
Differential trace
Covariance result (zoomed)

Introduction

- Suppose x is a sensitive intermediate variable in a block cipher.

Introduction

- Suppose x is a sensitive intermediate variable in a block cipher.
- Generate a random r, and process r and masked value

$$x' = x \oplus r$$

separately instead of x.

Introduction

- Suppose x is a sensitive intermediate variable in a block cipher.
- Generate a random r, and process r and masked value

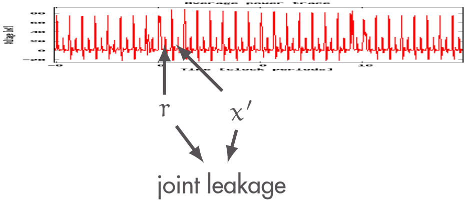
$$x' = x \oplus r$$

separately instead of x.

- r is random
  - $\Rightarrow$ x' is random
  - $\Rightarrow$ Power consumption of r or x' alone does not leak any information on x.

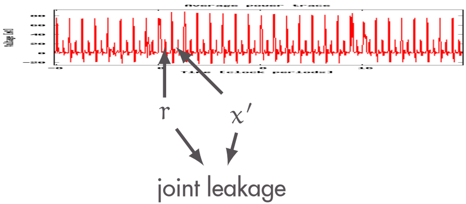
Introduction

- Second-order attacks
  - Two intermediate variables are probed.



  - More power traces and more complicated statistical techniques required but still practical.

- Second-order attacks
  - Two intermediate variables are probed.



  - More power traces and more complicated statistical techniques required but still practical.
- High-order attacks
  - $\mathtt{order}$ is the number of probed intermediate values.
  - The complexity grows exponentially as the $\mathtt{order}$ increases.

Outline

Background

- Published by National Institute of Standards and Technology (NIST) in 2001
- Substitution-permutation network based block cipher
- 128-bit (4*4 bytes) state block with three different key lengths

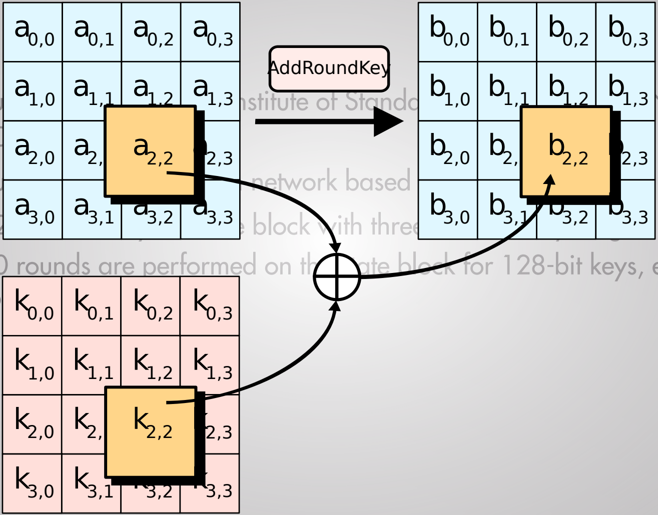
Background

- Published by National Institute of Standards and Technology (NIST) in 2001
- Substitution-permutation network based block cipher
- 128-bit (4*4 bytes) state block with three different key lengths
- 10 rounds are performed on the state block for 128-bit keys, each round contains:

Background

- Published by National Institute of Standards and Technology (NIST) in 2001
- Substitution-permutation network based block cipher
- 128-bit (4*4 bytes) state block with three different key lengths
- 10 rounds are performed on the state block for 128-bit keys, each round contains:
  - AddRoundKey

Background

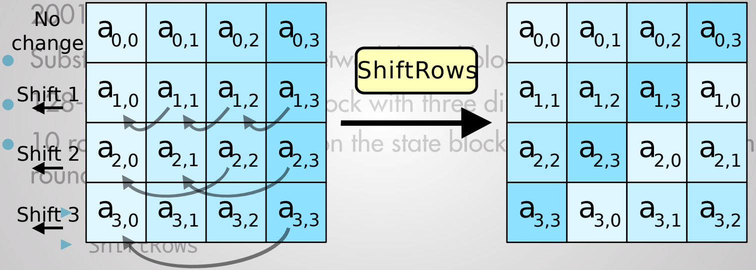


- Pu… nstitute of Standa… NIST) in 20…
- Su… network based
- 12… block with three
- 10 rounds are performed on th… ate block for 128-bit keys, each ro…

Background

- Published by National Institute of Standards and Technology (NIST) in 2001
- Substitution-permutation network based block cipher
- 128-bit (4*4 bytes) state block with three different key lengths
- 10 rounds are performed on the state block for 128-bit keys, each round contains:
  - `AddRoundKey`
  - `ShiftRows`

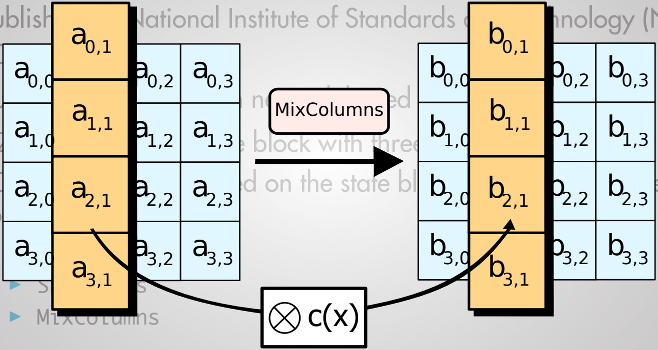
- Published by National Institute of Standards and Technology (NIST) in 2001



Background

- Published by National Institute of Standards and Technology (NIST) in 2001
- Substitution-permutation network based block cipher
- 128-bit (4*4 bytes) state block with three different key lengths
- 10 rounds are performed on the state block for 128-bit keys, each round contains:
  - `AddRoundKey`
  - `ShiftRows`
  - `MixColumns`

Background

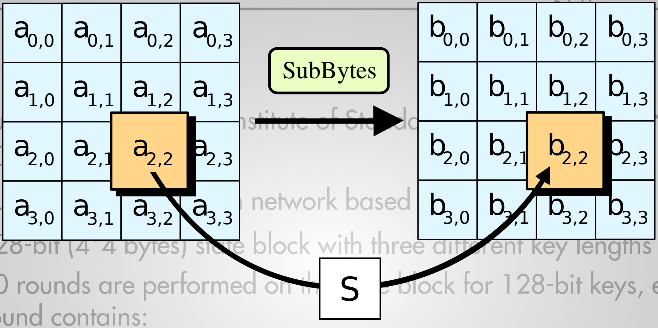

a0,0
a0,1
a0,2
a0,3
a1,0
a1,1
a1,2
a1,3
a2,0
a2,1
a2,2
a2,3
a3,0
a3,1
a3,2
a3,3
MixColumns
⊗ c(x)
b0,0
b0,1
b0,2
b0,3
b1,0
b1,1
b1,2
b1,3
b2,0
b2,1
b2,2
b2,3
b3,0
b3,1
b3,2
b3,3

Background

- Published by National Institute of Standards and Technology (NIST) in 2001
- Substitution-permutation network based block cipher
- 128-bit (4*4 bytes) state block with three different key lengths
- 10 rounds are performed on the state block for 128-bit keys, each round contains:
  - `AddRoundKey`
  - `ShiftRows`
  - `MixColumns`
  - `SubBytes`, also known as S-box, non-linear transformation

Background

Advanced Encryption Standard (AES)



- 128-bit (4·4 bytes) state block with three different key lengths
- 10 rounds are performed on the state block for 128-bit keys, each round contains:
  - AddRoundKey
  - ShiftRows
  - MixColumns
  - SubBytes, also known as S-box, non-linear transformation

S-box: a multiplicative inversion over $\mathbb{F}_{2^8}$ followed by an affine transformation.

Inversion: typically implemented via table look-up, but in our case: $x^{-1} = x^{254}$.

Background

- Intermediate value x is split into n shares: $x = x_1 \oplus \cdots \oplus x_n$ and these shares are manipulated separately.

Background

- Intermediate value x is split into n shares: $x = x_1 \oplus \cdots \oplus x_n$ and these shares are manipulated separately.
- Any subset of at most $n-1$ shares is independent of x
  - $\Rightarrow$ Any joint leakage of at most $n-1$ shares leaks nothing about x
  - $\Rightarrow$ Resistant against $(n-1)$-th order DPA attacks.

Background

- Intermediate value x is split into n shares: $x = x_1 \oplus \cdots \oplus x_n$ and these shares are manipulated separately.
- Any subset of at most $n-1$ shares is independent of x
  - $\Rightarrow$ Any joint leakage of at most $n-1$ shares leaks nothing about x
  - $\Rightarrow$ Resistant against $(n-1)$-th order DPA attacks.
- High-order masking countermeasures are practically sufficient for a certain order.

Background

- Intermediate value x is split into n shares: $x = x_1 \oplus \cdots \oplus x_n$ and these shares are manipulated separately.
- Any subset of at most $n-1$ shares is independent of x
  - $\Rightarrow$ Any joint leakage of at most $n-1$ shares leaks nothing about x
  - $\Rightarrow$ Resistant against $(n-1)$-th order DPA attacks.
- High-order masking countermeasures are practically sufficient for a certain order.
- Masking linear operation $f(\cdot)$ $f(x) = f(x_1) \oplus \cdots \oplus f(x_n)$.

Background

- Intermediate value x is split into n shares: $x = x_1 \oplus \cdots \oplus x_n$ and these shares are manipulated separately.
- Any subset of at most $n-1$ shares is independent of x
  - $\Rightarrow$ Any joint leakage of at most $n-1$ shares leaks nothing about x
  - $\Rightarrow$ Resistant against $(n-1)$-th order DPA attacks.
- High-order masking countermeasures are practically sufficient for a certain order.
- Masking linear operation $f(\cdot)$ $f(x) = f(x_1) \oplus \cdots \oplus f(x_n)$.
- Masking S-Boxes ?

Background

- Intermediate value x is split into n shares: $x = x_1 \oplus \cdots \oplus x_n$ and these shares are manipulated separately.
- Any subset of at most $n-1$ shares is independent of x
  - $\Rightarrow$ Any joint leakage of at most $n-1$ shares leaks nothing about x
  - $\Rightarrow$ Resistant against $(n-1)$-th order DPA attacks.
- High-order masking countermeasures are practically sufficient for a certain order.
- Masking linear operation $f(\cdot)$ $f(x) = f(x_1) \oplus \cdots \oplus f(x_n)$.
- Masking S-Boxes ? Not easy!!!

- Ishai-Sahai-Wagner Scheme [ISW03]
  - Describe how to transform a boolean circuit into a new circuit resistant against any t probes.
- Rivain-Prouff countermeasure [RP10]
  - Secure the inversion of S-box through exponentiation.
  - Secure the inversion of S-box over composite field [KHL11].
- Carlet `et al.` countermeasure (FSE12)
  - Extend [RP10] to arbitrary S-box

$$S(x) = \sum_{i=0}^{2^k-1} \alpha_i x^i$$

    over $\mathbb{F}_{2^k}$.
- Coron countermeasure (EUROCRYPT14)
  - Generalize the classic randomized table countermeasure.

Background

AES inversion (power function) $x \mapsto x^{254}$

- Secure exponentiation (inversion) consists of several secure multiplications and squarings.

Background

AES inversion (power function) $x \mapsto x^{254}$

- Secure exponentiation (inversion) consists of several secure multiplications and squarings.
- Secure squaring is easy.

AES inversion (power function) $x \mapsto x^{254}$

- Secure exponentiation (inversion) consists of several secure multiplications and squarings.
- Secure squaring is easy.
- Secure multiplication $z = xy$ is extended from [ISW03], i.e., recomputing

$$\bigoplus_{i=1}^{n} z_i = \left(\bigoplus_{i=1}^{n} x_i\right)\left(\bigoplus_{i=1}^{n} y_i\right) = \bigoplus_{1\leqslant i,j\leqslant n} x_i y_j$$

as

$$\begin{aligned} \bigoplus_i z_i &= \bigoplus_i \left(x_i y_i \oplus \bigoplus_{j<i} (x_i y_j \oplus x_j y_i)\right) \\ &= \bigoplus_i \left(\left(\bigoplus_{j>i} r_{i,j}\right) \oplus x_i y_i \oplus \bigoplus_{j<i} ((r_{j,i} \oplus x_i y_j) \oplus x_j y_i)\right). \end{aligned} \tag{1}$$

SecExp254 - masked exponentiation in $\mathbb{F}_{2^8}$ with n shares [RP10]

Input: shares x_i satisfying $x_1 \oplus \cdots \oplus x_n = x$
Output: shares y_i satisfying $y_1 \oplus \cdots \oplus y_n = x^{254}$

1: $(z_i)_i \leftarrow (x_i^2)_i$

SecExp254 - masked exponentiation in $\mathbb{F}_{2^8}$ with n shares [RP10]

Input: shares x_i satisfying $x_1 \oplus \cdots \oplus x_n = x$
Output: shares y_i satisfying $y_1 \oplus \cdots \oplus y_n = x^{254}$

1: $(z_i)_i \leftarrow (x_i^2)_i$ $\triangleright \bigoplus_i z_i = x^2$
2: RefreshMasks$((z_i)_i)$
3: $(y_i)_i \leftarrow$ SecMult$((z_i)_i, (x_i)_i)$ $\triangleright \bigoplus_i y_i = x^3$

SecExp254 - masked exponentiation in $\mathbb{F}_{2^8}$ with n shares [RP10]

Input: shares x_i satisfying $x_1 \oplus \cdots \oplus x_n = x$
Output: shares y_i satisfying $y_1 \oplus \cdots \oplus y_n = x^{254}$

1: $(z_i)_i \leftarrow (x_i^2)_i$ $\triangleright \bigoplus_i z_i = x^2$
2: RefreshMasks$((z_i)_i)$
3: $(y_i)_i \leftarrow$ SecMult$((z_i)_i, (x_i)_i)$ $\triangleright \bigoplus_i y_i = x^3$
4: $(w_i)_i \leftarrow (y_i^4)_i$ $\triangleright \bigoplus_i w_i = x^{12}$
5: RefreshMasks$((w_i)_i)$
6: $(y_i)_i \leftarrow$ SecMult$((y_i)_i, (w_i)_i)$ $\triangleright \bigoplus_i y_i = x^{15}$

SecExp254 - masked exponentiation in $\mathbb{F}_{2^8}$ with n shares [RP10]

Input: shares x_i satisfying $x_1 \oplus \cdots \oplus x_n = x$
Output: shares y_i satisfying $y_1 \oplus \cdots \oplus y_n = x^{254}$

1: $(z_i)_i \leftarrow (x_i^2)_i$ $\quad\triangleright \bigoplus_i z_i = x^2$
2: RefreshMasks$((z_i)_i)$
3: $(y_i)_i \leftarrow$ SecMult$((z_i)_i, (x_i)_i)$ $\quad\triangleright \bigoplus_i y_i = x^3$
4: $(w_i)_i \leftarrow (y_i^4)_i$ $\quad\triangleright \bigoplus_i w_i = x^{12}$
5: RefreshMasks$((w_i)_i)$
6: $(y_i)_i \leftarrow$ SecMult$((y_i)_i, (w_i)_i)$ $\quad\triangleright \bigoplus_i y_i = x^{15}$
7: $(y_i)_i \leftarrow (y_i^{16})_i$ $\quad\triangleright \bigoplus_i y_i = x^{240}$

SecExp254 - masked exponentiation in $\mathbb{F}_{2^8}$ with n shares [RP10]

Input: shares x_i satisfying $x_1 \oplus \cdots \oplus x_n = x$
Output: shares y_i satisfying $y_1 \oplus \cdots \oplus y_n = x^{254}$

1: $(z_i)_i \leftarrow (x_i^2)_i$ $\quad \triangleright \bigoplus_i z_i = x^2$
2: RefreshMasks$((z_i)_i)$
3: $(y_i)_i \leftarrow$ SecMult$((z_i)_i, (x_i)_i)$ $\quad \triangleright \bigoplus_i y_i = x^3$
4: $(w_i)_i \leftarrow (y_i^4)_i$ $\quad \triangleright \bigoplus_i w_i = x^{12}$
5: RefreshMasks$((w_i)_i)$
6: $(y_i)_i \leftarrow$ SecMult$((y_i)_i, (w_i)_i)$ $\quad \triangleright \bigoplus_i y_i = x^{15}$
7: $(y_i)_i \leftarrow (y_i^{16})_i$ $\quad \triangleright \bigoplus_i y_i = x^{240}$
8: $(y_i)_i \leftarrow$ SecMult$((y_i)_i, (w_i)_i)$ $\quad \triangleright \bigoplus_i y_i = x^{252}$

SecExp254 - masked exponentiation in $\mathbb{F}_{2^8}$ with n shares [RP10]

Input: shares x_i satisfying $x_1 \oplus \cdots \oplus x_n = x$
Output: shares y_i satisfying $y_1 \oplus \cdots \oplus y_n = x^{254}$

1: $(z_i)_i \leftarrow (x_i^2)_i$ $\triangleright \bigoplus_i z_i = x^2$
2: RefreshMasks$((z_i)_i)$
3: $(y_i)_i \leftarrow$ SecMult$((z_i)_i, (x_i)_i)$ $\triangleright \bigoplus_i y_i = x^3$
4: $(w_i)_i \leftarrow (y_i^4)_i$ $\triangleright \bigoplus_i w_i = x^{12}$
5: RefreshMasks$((w_i)_i)$
6: $(y_i)_i \leftarrow$ SecMult$((y_i)_i, (w_i)_i)$ $\triangleright \bigoplus_i y_i = x^{15}$
7: $(y_i)_i \leftarrow (y_i^{16})_i$ $\triangleright \bigoplus_i y_i = x^{240}$
8: $(y_i)_i \leftarrow$ SecMult$((y_i)_i, (w_i)_i)$ $\triangleright \bigoplus_i y_i = x^{252}$
9: $(y_i)_i \leftarrow$ SecMult$((y_i)_i, (z_i)_i)$ $\triangleright \bigoplus_i y_i = x^{254}$

Background

1. $(z_i)_i \leftarrow (x_i^2)_i$
2. RefreshMasks$((z_i)_i)$
3. $(y_i)_i \leftarrow$ SecMult$((x_i)_i, (z_i)_i)$
4. $(w_i)_i \leftarrow (y_i^4)_i$
5. RefreshMasks$((w_i)_i)$
6. $(y_i)_i \leftarrow$ SecMult$((y_i)_i, (w_i)_i)$
7. $(y_i)_i \leftarrow (y_i^{16})_i$
8. $(y_i)_i \leftarrow$ SecMult$((y_i)_i, (w_i)_i)$
9. $(y_i)_i \leftarrow$ SecMult$((y_i)_i, (z_i)_i)$

1. $(z_i)_i \leftarrow (x_i^2)_i$
2. $\mathsf{RefreshMasks}((z_i)_i)$
3. $(y_i)_i \leftarrow \mathsf{SecMult}((x_i)_i, (z_i)_i)$
4. $(w_i)_i \leftarrow (y_i^4)_i$
5. $\mathsf{RefreshMasks}((w_i)_i)$
6. $(y_i)_i \leftarrow \mathsf{SecMult}((y_i)_i, (w_i)_i)$
7. $(y_i)_i \leftarrow (y_i^{16})_i$
8. $(y_i)_i \leftarrow \mathsf{SecMult}((y_i)_i, (w_i)_i)$
9. $(y_i)_i \leftarrow \mathsf{SecMult}((y_i)_i, (z_i)_i)$

1. $(z_i)_i \leftarrow (x_i^2)_i$
2. $\mathsf{RefreshMasks}((z_i)_i)$
3. $(y_i)_i \leftarrow \mathsf{SecMult}((x_i)_i, (z_i)_i)$
4. $(w_i)_i \leftarrow (y_i^4)_i$
5. $\mathsf{RefreshMasks}((w_i)_i)$
6. $(y_i)_i \leftarrow \mathsf{SecMult}((y_i)_i, (w_i)_i)$
7. $(y_i)_i \leftarrow (y_i^{16})_i$
8. $(y_i)_i \leftarrow \mathsf{SecMult}((y_i)_i, (w_i)_i)$
9. $(y_i)_i \leftarrow \mathsf{SecMult}((y_i)_i, (z_i)_i)$

Background

1. $(z_i)_i \leftarrow (x_i^2)_i$
2. $\texttt{RefreshMasks}((z_i)_i)$
3. $(y_i)_i \leftarrow \texttt{SecMult}((x_i)_i, (z_i)_i)$
4. $(w_i)_i \leftarrow (y_i^4)_i$
5. $\texttt{RefreshMasks}((w_i)_i)$
6. $(y_i)_i \leftarrow \texttt{SecMult}((y_i)_i, (w_i)_i)$
7. $(y_i)_i \leftarrow (y_i^{16})_i$
8. $(y_i)_i \leftarrow \texttt{SecMult}((y_i)_i, (w_i)_i)$
9. $(y_i)_i \leftarrow \texttt{SecMult}((y_i)_i, (z_i)_i)$

- Vulnerable to $(\lfloor n/2 \rfloor + 1)$-th order attacks due to the integration with `RefreshMasks`.

1. $(z_i)_i \leftarrow (x_i^2)_i$
2. $\texttt{RefreshMasks}((z_i)_i)$
3. $(y_i)_i \leftarrow \texttt{SecMult}((x_i)_i, (z_i)_i)$
4. $(w_i)_i \leftarrow (y_i^4)_i$
5. $\texttt{RefreshMasks}((w_i)_i)$
6. $(y_i)_i \leftarrow \texttt{SecMult}((y_i)_i, (w_i)_i)$
7. $(y_i)_i \leftarrow (y_i^{16})_i$
8. $(y_i)_i \leftarrow \texttt{SecMult}((y_i)_i, (w_i)_i)$
9. $(y_i)_i \leftarrow \texttt{SecMult}((y_i)_i, (z_i)_i)$

- Vulnerable to $(\lfloor n/2 \rfloor + 1)$-th order attacks due to the integration with `RefreshMasks`.

- Solution: secure the multiplication: $h(x) = x \cdot g(x)$, where $g(x) = x^{2k}$.

1. $(z_i)_i \leftarrow (x_i^2)_i$
2. ~~RefreshMasks$((z_i)_i)$~~
3. $(y_i)_i \leftarrow$ $\boxed{\texttt{SecH}}$$((x_i)_i, (z_i)_i)$
4. $(w_i)_i \leftarrow (y_i^4)_i$
5. ~~RefreshMasks$((w_i)_i)$~~
6. $(y_i)_i \leftarrow$ $\boxed{\texttt{SecH}}$$((y_i)_i, (w_i)_i)$
7. $(y_i)_i \leftarrow (y_i^{16})_i$
8. $(y_i)_i \leftarrow$ SecMult$((y_i)_i, (w_i)_i)$
9. $(y_i)_i \leftarrow$ SecMult$((y_i)_i, (z_i)_i)$

- Vulnerable to $(\lfloor n/2 \rfloor + 1)$-th order attacks due to the integration with `RefreshMasks`.

- Solution: secure the multiplication: $h(x) = x \cdot g(x)$, where $g(x) = x^{2k}$.

1. $(z_i)_i \leftarrow (x_i^2)_i$
2. ~~RefreshMasks$((z_i)_i)$~~
3. $(y_i)_i \leftarrow$ SecH$((x_i)_i, (z_i)_i)$
4. $(w_i)_i \leftarrow (y_i^4)_i$
5. ~~RefreshMasks$((w_i)_i)$~~
6. $(y_i)_i \leftarrow$ SecH$((y_i)_i, (w_i)_i)$
7. $(y_i)_i \leftarrow (y_i^{16})_i$
8. $(y_i)_i \leftarrow$ SecMult$((y_i)_i, (w_i)_i)$
9. $(y_i)_i \leftarrow$ SecMult$((y_i)_i, (z_i)_i)$

- Vulnerable to $(\lfloor n/2 \rfloor + 1)$-th order attacks due to the integration with `RefreshMasks`.

- Solution: secure the multiplication: $h(x) = x \cdot g(x)$, where $g(x) = x^{2k}$.
- Suppose $f(x_i, x_j) = x_i \cdot g(x_j) \oplus x_j \cdot g(x_i)$

Background

1. $(z_i)_i \leftarrow (x_i^2)_i$
2. ~~RefreshMasks$((z_i)_i)$~~
3. $(y_i)_i \leftarrow$ SecH$((x_i)_i, (z_i)_i)$
4. $(w_i)_i \leftarrow (y_i^4)_i$
5. ~~RefreshMasks$((w_i)_i)$~~
6. $(y_i)_i \leftarrow$ SecH$((y_i)_i, (w_i)_i)$
7. $(y_i)_i \leftarrow (y_i^{16})_i$
8. $(y_i)_i \leftarrow$ SecMult$((y_i)_i, (w_i)_i)$
9. $(y_i)_i \leftarrow$ SecMult$((y_i)_i, (z_i)_i)$

- Vulnerable to $(\lfloor n/2 \rfloor + 1)$-th order attacks due to the integration with `RefreshMasks`.

- Solution: secure the multiplication: $h(x) = x \cdot g(x)$, where $g(x) = x^{2k}$.
- Suppose $f(x_i, x_j) = x_i \cdot g(x_j) \oplus x_j \cdot g(x_i)$
- By the property of $f(\cdot, \cdot)$ that $f(x_i, x_j) = f(x_i, r) \oplus f(x_i, x_j \oplus r)$

1. $(z_i)_i \leftarrow (x_i^2)_i$
2. ~~$\texttt{RefreshMasks}((z_i)_i)$~~
3. $(y_i)_i \leftarrow \texttt{SecH}((x_i)_i, (z_i)_i)$
4. $(w_i)_i \leftarrow (y_i^4)_i$
5. ~~$\texttt{RefreshMasks}((w_i)_i)$~~
6. $(y_i)_i \leftarrow \texttt{SecH}((y_i)_i, (w_i)_i)$
7. $(y_i)_i \leftarrow (y_i^{16})_i$
8. $(y_i)_i \leftarrow \texttt{SecMult}((y_i)_i, (w_i)_i)$
9. $(y_i)_i \leftarrow \texttt{SecMult}((y_i)_i, (z_i)_i)$

- Vulnerable to $(\lfloor n/2 \rfloor + 1)$-th order attacks due to the integration with `RefreshMasks`.

- Solution: secure the multiplication: $h(x) = x \cdot g(x)$, where $g(x) = x^{2k}$.
- Suppose $f(x_i, x_j) = x_i \cdot g(x_j) \oplus x_j \cdot g(x_i)$
- By the property of $f(\cdot, \cdot)$ that $f(x_i, x_j) = f(x_i, r) \oplus f(x_i, x_j \oplus r)$
- Equation 1 equals to

$$\begin{aligned}
\bigoplus_i z_i = & \bigoplus_i \left(\left(\bigoplus_{j>i} r_{i,j} \right) \oplus x_i y_i \oplus \bigoplus_{j<i} \left(r_{j,i} \oplus f(x_i, x_j) \right) \right) \\
= & \bigoplus_i \left(\left(\bigoplus_{j>i} r_{i,j} \right) \oplus x_i y_i \oplus \bigoplus_{j<i} \left(r_{j,i} \oplus f(x_i, r'_{j,i}) \oplus f(x_i, x_j \oplus r'_{j,i}) \right) \right),
\end{aligned}$$

if $y_i = g(x_i)$.

Outline

- Introduction
  - Differential Power Analysis
  - Masking Countermeasures
  - High-Order DPA Attacks
- Background
  - Advanced Encryption Standard
  - High-Order Masking
  - Rivain-Prouff Countermeasure
- Implementation
  - ARM NEON
  - Performance-Critical Analysis
  - Implementation of Secure Field Multiplication
- Results and Comparison
- Conclusion

Implementation

- ARM is a family of embedded processors
  - Low-cost, high-performance and energy-efficiency
  - Applications: smartphones, tablets, digital camera, etc.

Implementation

- ARM is a family of embedded processors
  - Low-cost, high-performance and energy-efficiency
  - Applications: smartphones, tablets, digital camera, etc.
- NEON is an advanced SIMD extension on modern ARM processors

Implem… …ARM NEON

- ARM is …
  - Lov…
  - Ap…
- NEON i… … processors

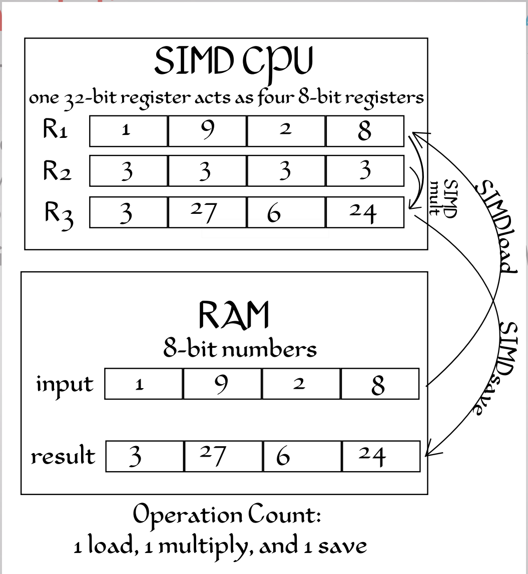


Figure: SIMD Example

Implementation

- ARM is a family of embedded processors
  - Low-cost, high-performance and energy-efficiency
  - Applications: smartphones, tablets, digital camera, etc.
- NEON is an advanced SIMD extension on modern ARM processors
  - Accelerate multimedia and signal processing

Implementation

- ARM is a family of embedded processors
  - Low-cost, high-performance and energy-efficiency
  - Applications: smartphones, tablets, digital camera, etc.
- NEON is an advanced SIMD extension on modern ARM processors
  - Accelerate multimedia and signal processing
  - Registers: thirty-two 64-bit registers (can also be viewed as sixteen 128-bit register)

Implementation

- ARM is a family of embedded processors
  - Low-cost, high-performance and energy-efficiency
  - Applications: smartphones, tablets, digital camera, etc.
- NEON is an advanced SIMD extension on modern ARM processors
  - Accelerate multimedia and signal processing
  - Registers: thirty-two 64-bit registers (can also be viewed as sixteen 128-bit register)
  - Data Type: 8-, 16-, 32- and 64-bit (signed/unsigned) integers and 8- and 16-bit polynomial

Implementation

- ARM is a family of embedded processors
  - Low-cost, high-performance and energy-efficiency
  - Applications: smartphones, tablets, digital camera, etc.
- NEON is an advanced SIMD extension on modern ARM processors
  - Accelerate multimedia and signal processing
  - Registers: thirty-two 64-bit registers (can also be viewed as sixteen 128-bit register)
  - Data Type: 8-, 16-, 32- and 64-bit (signed/unsigned) integers and 8- and 16-bit polynomial
  - Arithmetic operations, boolean operations and others

Implementation

- ARM is a family of embedded processors
  - Low-cost, high-performance and energy-efficiency
  - Applications: smartphones, tablets, digital camera, etc.
- NEON is an advanced SIMD extension on modern ARM processors
  - Accelerate multimedia and signal processing
  - Registers: thirty-two 64-bit registers (can also be viewed as sixteen 128-bit register)
  - Data Type: 8-, 16-, 32- and 64-bit (signed/unsigned) integers and 8- and 16-bit polynomial
  - Arithmetic operations, boolean operations and others
  - Featured instruction:
    - `VMULL.P8`
    - `VTBL.8`

Operations	Field Multiplication	Random Bits	XOR	Momeory
`SecSqur`	n	0	0	$2n$
`SecPow4`	$2n$	0	0	$2n$
`SecPow16`	$4n$	0	0	$2n$
`SecMult`	n^2	$(n^2-n)/2$	$2(n^2-n)$	$2n+\mathcal{O}(1)$
`SecH`	$(n^2-n)(m+2)+n$	n^2-n	$7(n^2-n)/2$	$3n+\mathcal{O}(1)$
`SecExp254'`	$9n^2+2n$	$3(n^2-n)$	$11(n^2-n)$	$4n+\mathcal{O}(1)$

Table: Compleixty of masked algorithms for S-box with n shares, where m is the number of field multiplication in $h(\cdot)$.

Operations	Field Multiplication	Random Bits	XOR	Momeory
`SecSqur`	n	0	0	$2n$
`SecPow4`	$2n$	0	0	$2n$
`SecPow16`	$4n$	0	0	$2n$
`SecMult`	n^2	$(n^2-n)/2$	$2(n^2-n)$	$2n+\mathcal{O}(1)$
`SecH`	$(n^2-n)(m+2)+n$	n^2-n	$7(n^2-n)/2$	$3n+\mathcal{O}(1)$
`SecExp254'`	$9n^2+2n$	$3(n^2-n)$	$11(n^2-n)$	$4n+\mathcal{O}(1)$

Table: Compleixty of masked algorithms for S-box with n shares, where m is the number of field multiplication in $h(\cdot)$.

- Performance-critical parts:
  - Field Multiplication
  - Random bits generation

Implementation

- Designed to optimize the modular reduction $r = a \bmod n$, where a, n are integers and $a < n^2$.

Implementation

- Designed to optimize the modular reduction $r = a \bmod n$, where a, n are integers and $a < n^2$.
- Adapted to polynomials [Dhe03]
  - Suppose $U(x)$, $Q(x)$, $N(x)$ and $Z(x)$ are polynomial over $\mathbb{F}_q$, and $U(x) = Q(x)N(x) + Z(x)$
  - $\lfloor A(x)/B(x) \rfloor$ stands for the quotient of $A(x)/B(x)$, ignoring the reminder
  - Quotient evaluation

$$Q(x) = \left\lfloor \frac{U(x)}{N(x)} \right\rfloor = \left\lfloor \frac{\left\lfloor \frac{U(x)}{x^p} \right\rfloor \left\lfloor \frac{x^{p+\beta}}{N(x)} \right\rfloor}{x^\beta} \right\rfloor = \left\lfloor \frac{T(x)R(x)}{x^\beta} \right\rfloor,$$

    where $p = \deg(N(x))$, $\beta \geqslant \deg(U(x)/x^p)$
  - The reminder $Z(x) = U(x) - Q(x)N(x)$.

Implementation

Input: polynomials $A(x)$, $B(x)$ and $N(x)$ in $\mathbb{F}_{2^8}$, where $N(x) = x^8 + x^4 + x^3 + x + 1$
Output: polynomial $Z(x) = A(x) \cdot B(x) \bmod N(x)$
Pre-computation:

1: $p \leftarrow \deg(N(x))$ $\quad \triangleright\ p = 8$
2: $\alpha \leftarrow 2 * (p - 1)$ $\quad \triangleright\ \alpha = 14$
3: $\beta \geqslant \alpha - p$ $\quad \triangleright\ \beta \geqslant 6$
4: $R(x) \leftarrow \lfloor \frac{x^{p+\beta}}{N(x)} \rfloor$ $\quad \triangleright\ R(x) = x^6 + x^2 + x$ if $\beta = 6$

Implementation

Input: polynomials $A(x)$, $B(x)$ and $N(x)$ in $\mathbb{F}_{2^8}$, where $N(x) = x^8 + x^4 + x^3 + x + 1$
Output: polynomial $Z(x) = A(x) \cdot B(x) \bmod N(x)$
Pre-computation:

1: $p \leftarrow deg(N(x))$ $\triangleright\ p = 8$
2: $\alpha \leftarrow 2 * (p - 1)$ $\triangleright\ \alpha = 14$
3: $\beta \geqslant \alpha - p$ $\triangleright\ \beta \geqslant 6$
4: $R(x) \leftarrow \lfloor \frac{x^{p+\beta}}{N(x)} \rfloor$ $\triangleright\ R(x) = x^6 + x^2 + x$ if $\beta = 6$

Multiplication with Barrett modular reduction:

1: $U(x) \leftarrow A(x) \cdot B(x)$ $\triangleright\ deg(U(x)) \leqslant 14$
2: $T(x) \leftarrow \lfloor \frac{U(x)}{x^p} \rfloor$ $\triangleright\ deg(T(x)) \leqslant 6$
3: $S(x) \leftarrow T(x) \cdot R(x)$ $\triangleright\ deg(S(x)) \leqslant \beta + 6$
4: $Q(x) \leftarrow \lfloor \frac{S(x)}{x^\beta} \rfloor$ $\triangleright\ deg(Q(x)) \leqslant 6$
5: $V(x) \leftarrow Q(x) \cdot N(x)$ $\triangleright\ deg(V(x)) \leqslant 14$
6: $Z(x) \leftarrow U(x) + V(x)$

```
fmult:      /*uint8x16_t fmult(uint8x16_t a, uint8x16_t b)*/
```

Implementation


```
fmult:          /*uint8x16_t fmult(uint8x16_t a, uint8x16_t b)*/
    VMULL.P8   Q2,D1,D3          1. U(x) = A(x) * B(x)
    VMULL.P8   Q1,D0,D2
    VMOVN.I16 D0,Q1
    VMOVN.I16 D1,Q2
```

Implementation


```
fmult:
    VMULL.P8   Q2,D1,D3
    VMULL.P8   Q1,D0,D2
    VMOVN.I16  D0,Q1
    VMOVN.I16  D1,Q2
    VSHRN.U16  D2,Q1,#+8
    VSHRN.U16  D3,Q2,#+8
```

1. $U(x) = A(x) * B(x)$

2. $T(x) = U(x) / x^8$

Implementation


```
fmult:
    VMULL.P8   Q2,D1,D3          1. U(x) = A(x) * B(x)
    VMULL.P8   Q1,D0,D2
    VMOVN.I16  D0,Q1
    VMOVN.I16  D1,Q2
    VSHRN.U16  D2,Q1,#+8         2. T(x) = U(x) / x^8
    VSHRN.U16  D3,Q2,#+8
    VMOV.U8    D7,#+70           3. S(x) = T(x) * R(x)
    VMULL.P8   Q2,D2,D7          4. Q(x) = S(x) / x^6
    VSHRN.U16  D2,Q2,#+6
    VMULL.P8   Q2,D3,D7
    VSHRN.U16  D3,Q2,#+6
```

Implementation


```
fmult:
    VMULL.P8   Q2,D1,D3          1. U(x) = A(x) * B(x)
    VMULL.P8   Q1,D0,D2
    VMOVN.I16  D0,Q1
    VMOVN.I16  D1,Q2
    VSHRN.U16  D2,Q1,#+8         2. T(x) = U(x) / x^8
    VSHRN.U16  D3,Q2,#+8
    VMOV.U8    D7,#+70           3. S(x) = T(x) * R(x)
    VMULL.P8   Q2,D2,D7          4. Q(x) = S(x) / x^6
    VSHRN.U16  D2,Q2,#+6
    VMULL.P8   Q2,D3,D7
    VSHRN.U16  D3,Q2,#+6
    VMOV.U8    D2,#0x1B          5. V(x) = Q(x) * N(x)
    VMULL.P8   Q1,Q2,Q1
```

Implementation


```
fmult:
    VMULL.P8   Q2,D1,D3        1. U(x) = A(x) * B(x)
    VMULL.P8   Q1,D0,D2
    VMOVN.I16  D0,Q1
    VMOVN.I16  D1,Q2
    VSHRN.U16  D2,Q1,#+8       2. T(x) = U(x) / x^8
    VSHRN.U16  D3,Q2,#+8
    VMOV.U8    D7,#+70         3. S(x) = T(x) * R(x)
    VMULL.P8   Q2,D2,D7        4. Q(x) = S(x) / x^6
    VSHRN.U16  D2,Q2,#+6
    VMULL.P8   Q2,D3,D7
    VSHRN.U16  D3,Q2,#+6
    VMOV.U8    D2,#0x1B        5. V(x) = Q(x) * N(x)
    VMULL.P8   Q1,Q2,Q1
    VEOR       Q0,Q1,Q0        6. Z(x) = U(x) + V(x)
    BX         LR
```

```
void sec_fmult(uint8x16_t c[],
uint8x16_t a[], uint8x16_t b[],
int n) {
  int i, j;
  uint8x16_t s, t;

  for (i = 0; i < n; i++)
    c[i] = fmult(a[i], b[i]);
  for (i = 0; i < n; i++)
    for (j = i+1; j < n; j++) {
      s = rand_uint8x16();
      c[i] = veorq_u8(c[i], s);
      t = fmult(a[i], b[j]);
      s = veorq_u8(s, t);
      t = fmult(a[j], b[i]);
      s = veorq_u8(s, t);
      c[j] = veorq_u8(c[j], s);
    }
}
```

```
void sec_h(uint8x16_t y[],
uint8x16_t x[], uint8x16_t gx[],
uint8x16_t (g_call)(uint8x16_t),int n) {
            ⋮
  for (…)
    for (…) {
            ⋮
      t = g_call(r01);
      t = fmult(x[i], t);
      r1 = veorq_u8(r00, t);
      t = fmult(r01, gx[i]);
      r1 = veorq_u8(r1, t);
      s = veorq_u8(x[j], r01);
      t = g_call(s);
      t = fmult(x[i], t);
      r1 = veorq_u8(t, r1);
      t = fmult(gx[i], s);
      r1 = veorq_u8(t, r1);
      y[j] = veorq_u8(y[j], r1);
    }
}
```

Implementation

```
void sec_fmult(uint8x16_t c[],
uint8x16_t a[], uint8x16_t b[],
int n) {
  int i, j;
  uint8x16_t s, t;

  for (i = 0; i < n; i++)
    c[i] = fmult(a[i], b[i]);
  for (i = 0; i < n; i++)
    for (j = i+1; j < n; j++) {
      s = rand_uint8x16();
      c[i] = veorq_u8(c[i], s);
      t = fmult(a[i], b[j]);
      s = veorq_u8(s, t);
      t = fmult(a[j], b[i]);
      s = veorq_u8(s, t);
      c[j] = veorq_u8(c[j], s);
    }
}
```

```
void sec_h(uint8x16_t y[],
uint8x16_t x[], uint8x16_t gx[],
uint8x16_t (g_call)(uint8x16_t),int n) {
            ⋮
  for (…)
    for (…) {
            ⋮
      t = g_call(r01);
      t = fmult(x[i], t);
      r1 = veorq_u8(r00, t);
      t = fmult(r01, gx[i]);
      r1 = veorq_u8(r1, t);
      s = veorq_u8(x[j], r01);
      t = g_call(s);
      t = fmult(x[i], t);
      r1 = veorq_u8(t, r1);
      t = fmult(gx[i], s);
      r1 = veorq_u8(t, r1);
      y[j] = veorq_u8(y[j], r1);
    }
}
```

Implementation


```
void sec_fmult(uint8x16_t c[],
uint8x16_t a[], uint8x16_t b[],
int n) {
  int i, j;
  uint8x16_t s, t;

  for (i = 0; i < n; i++)
    c[i] = fmult(a[i], b[i]);
  for (i = 0; i < n; i++)
    for (j = i+1; j < n; j++) {
      s = rand_uint8x16();
      c[i] = veorq_u8(c[i], s);
      t = fmult(a[i], b[j]);
      s = veorq_u8(s, t);
      t = fmult(a[j], b[i]);
      s = veorq_u8(s, t);
      c[j] = veorq_u8(c[j], s);
    }
}
```

```
void sec_h(uint8x16_t y[],
uint8x16_t x[], uint8x16_t gx[],
uint8x16_t (g_call)(uint8x16_t),int n) {
        ⋮
  for (…)
    for (…) {
        ⋮
      t = g_call(r01);
      t = fmult(x[i], t);
      r1 = veorq_u8(r00, t);
      t = fmult(r01, gx[i]);
      r1 = veorq_u8(r1, t);
      s = veorq_u8(x[j], r01);
      t = g_call(s);
      t = fmult(x[i], t);
      r1 = veorq_u8(t, r1);
      t = fmult(gx[i], s);
      r1 = veorq_u8(t, r1);
      y[j] = veorq_u8(y[j], r1);
    }
}
```

Implementation

- [KHL11] is vulnerable to the same attack on [RP10]
- We propose a new secure inversion algorithm

SecInv4 - masked exponentiation in $\mathbb{F}_{2^4}$ with n shares

Input: shares x_i satisfying $x_1 \oplus \cdots \oplus x_n = x$
Output: shares y_i satisfying $y_1 \oplus \cdots \oplus y_n = x^{14}$

1: $(w_i)_i \leftarrow (x_i^2)_i$ $\triangleright \bigoplus_i w_i = x^2$
2: $(z_i)_i \leftarrow \mathsf{SecH}((x_i)_i, (w_i)_i)$ $\triangleright \bigoplus_i z_i = x^3$
3: $(z_i)_i \leftarrow (z_i^4)_i$ $\triangleright \bigoplus_i z_i = x^{12}$
4: $(y_i)_i \leftarrow \mathsf{SecMult}((z_i)_i, (w_i)_i)$ $\triangleright \bigoplus_i y_i = x^{14}$

Outline

Introduction
- Differential Power Analysis
- Masking Countermeasures
- High-Order DPA Attacks

Background
- Advanced Encryption Standard
- High-Order Masking
- Rivain-Prouff Countermeasure

Implementation
- ARM NEON
- Performance-Critical Analysis
- Implementation of Secure Field Multiplication

Results and Comparison

Conclusion

Results

Peformance Metrics	#instructions
Field Multiplication	15
Random Bits Generation - `xorshift96`	15
XOR	1
Secure AddRoundKey	n
Secure ShiftRows	$4n$
Secure MixColumns	$13n$
Secure Affine Transformation	$18n$
Secure Exp254	$191n^2 - 26n$

Table: The number of instructions comsumed by vector implementation of each elements, where n is the number of shares

Comparison

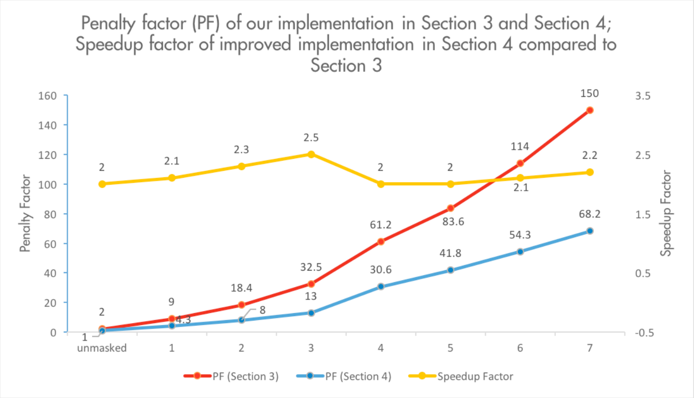


Figure: Comparison with baseline implementation

Comparison

Method	Platform	First-order	Second-order	Third-order	Fourth-order
CHES'10 [RP10]	8-bit 8051	65	132	235	-
CHES'11 [KHL11]	8-bit AVR	-	22	39	-
Coron [Cor14]	1.86 GHz Intel	439	1205	2411	4003
Ours (Section 3)	32-bit ARM	9	19	32	60
Ours (Section 4)	32-bit ARM	4	8	13	31

Table: Penalty factor in different implementations

Outline

Conclusion

- The performance-critical parts are field multiplication and random bits generation.

Conclusion

- The performance-critical parts are field multiplication and random bits generation.
- We utilize `vmull.p8` instruction and Barrett Reduction to optimize field multiplication, which only takes 15 instructions.

Conclusion

- The performance-critical parts are field multiplication and random bits generation.
- We utilize `vmull.p8` instruction and Barrett Reduction to optimize field multiplication, which only takes 15 instructions.
- We further improve our performance by using composite field $\mathrm{GF}(2^8) \triangleq \mathrm{GF}((2^4)^2)$.

Conclusion

- The performance-critical parts are field multiplication and random bits generation.
- We utilize `vmull.p8` instruction and Barrett Reduction to optimize field multiplication, which only takes 15 instructions.
- We further improve our performance by using composite field $\mathrm{GF}(2^8) \triangleq \mathrm{GF}((2^4)^2)$.
- Our implementation achieve a not bad penalty factor, hence, they are deployable in practice.

Conclusion

- The performance-critical parts are field multiplication and random bits generation.
- We utilize `vmull.p8` instruction and Barrett Reduction to optimize field multiplication, which only takes 15 instructions.
- We further improve our performance by using composite field $\mathrm{GF}(2^8) \triangleq \mathrm{GF}((2^4)^2)$.
- Our implementation achieve a not bad penalty factor, hence, they are deployable in practice.

Thank You!

Conclusion

- The performance-critical parts are field multiplication and random bits generation.
- We utilize `vmull.p8` instruction and Barrett Reduction to optimize field multiplication, which only takes 15 instructions.
- We further improve our performance by using composite field $GF(2^8) \triangleq GF((2^4)^2)$.
- Our implementation achieve a not bad penalty factor, hence, they are deployable in practice.

Question?