Implementing GCM on ARMv8

Conrado P. L. Gouvêa Julio López

KRYPTUS Information Security Solutions University of Campinas

CT-RSA 2015

Cryptography Performance Matters

Sections ☰ The Washington Post Search Sign In Subscribe

The Switch

Newest Androids will join iPhones in offering default encryption, blocking police

By Craig Timberg September 18, 2014 Follow @craigtimberg

Get the The Switchboard Newsletter

Free daily updates delivered just for you.

Cryptography Performance Matters

REVIEWS PHONES TABLETS APPS DEVICES DE

Android 5.0 encryption brings storage performance issues

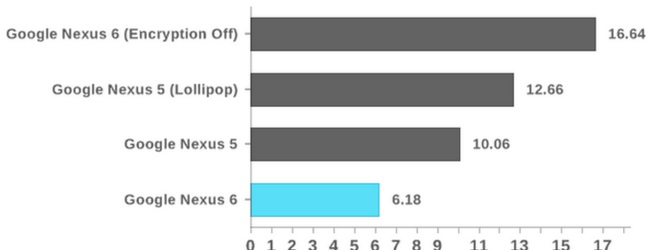
2

November 21, 2014

Like 244 Tweet 35 Compartilhar 29 submit

Internal NAND - Random Read

4KB Random Reads in MB/s - Higher is Better



Cryptography Performance Matters

PHANDROID DEALS FORUM PHONES TABLETS APPS MORE

CURRENTLY HOT: HTC One M9 Samsung Galaxy S6 Galaxy S6 Edge LG G4 Nexus 6 Wallpaper Podcast

How to give your Nexus 6 a huge speed boost by disabling device encryption

Nov 20th 2014 by Jeff McIntire 13 57

Android Forums

Cryptography Performance Matters

THE VERGE

TRENDING NOW
The loneliest roast: Justin Bieber does public penance on Comedy Central

30 NEW ARTICLES

LOG IN | SIGN UP LONGFORM VIDEO REVIEWS TECH SCIENCE ENTERTAINMENT DESIGN BUSINESS US & WORLD FORUMS

PREVIOUS STORY
The Ellen Pao trial is spilling Silicon Valley secrets

NEXT STORY
Ferguson police showed patterns of racial bias for years, says Justice...

GOOGLE MOBILE TECH

83 COMMENTS

Android Lollipop won't use default disk encryption due to performance issues

By Russell Brandom on March 3, 2015 03:21 pm Email @russellbrandom

This Work

- Efficient software implementation of GCM over AES for ARMv8
- Resistance to timing attacks
- Authenticated Encryption
  - Combine encryption and authentication in a single scheme, preventing mistakes

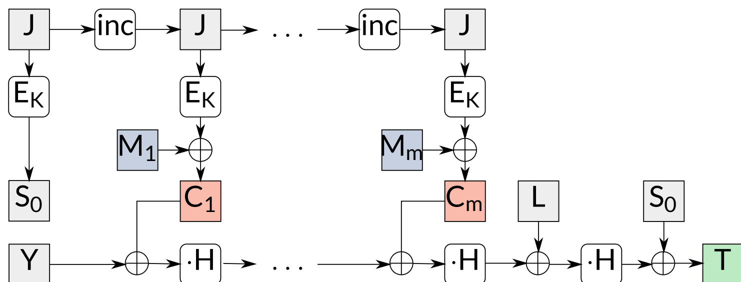
ARM Architecture

- Used by 95% of smartphones
- ARMv7: 32-bit, SIMD instruction set (NEON)
- ARMv8: 32-bit mode (AArch32), 64-bit mode (AArch64)

Galois/Counter Mode (GCM)

- McGrew and Viega, 2005
- Authenticated Encryption
  - Input: nonce, plaintext, additional data
  - Output: ciphertext, authentication tag
- Used in TLS, IPSec, SSH, NIST SP 800-38D
- Works with any 128-bit block cipher; used mostly with AES

Galois/Counter Mode (GCM)



- Uses CTR mode for encryption and defines the GHASH function for authentication
- GHASH uses binary field multiplication over $\mathbb{F}_{2^{128}}$

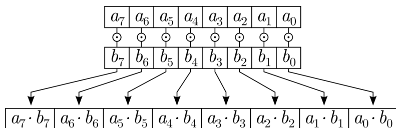
GCM Bit Order

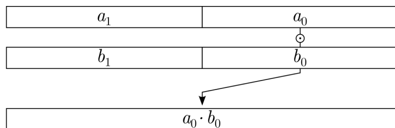
$$a(z) = 1$$

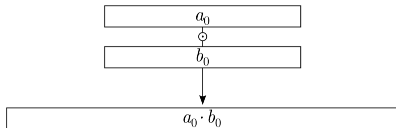
```
80 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

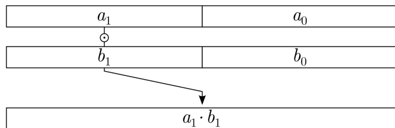
- Can’t shift words — breaks the bit order
- Workarounds:
  - Reverse bits in each byte, carry out computations, reverse again at the end
  - (Gueron and Kounavis 2010) Reverse the bytes in the vector, compute using “reverse modular reduction”, reverse again at the end

ARM Binary Polynomial Multiplication Support









- ARMv7
  - `VMULL.P8`
- ARMv8
  - AArch32: `VMULL.P64`
  - AArch64: `PMULL`, `PMULL2`

ARM AES Support

```
1:  aese.16b t0, k00
2:  aesmc.16b t0, t0
3:  aese.16b t0, k01
4:  aesmc.16b t0, t0
5:  aese.16b t0, k02
6:  aesmc.16b t0, t0
7:  aese.16b t0, k03
8:  aesmc.16b t0, t0
9:  aese.16b t0, k04
10: aesmc.16b t0, t0
11: ...
```

- Advanced Encryption Standard
- AES instructions in ARMv8 (both AArch32 and AArch64)
- `AESE` (AddRoundKey, SubBytes, ShiftRows)
- `AESMC` (MixColumns)
- `AESD`, `AESIMC` (Decryption)

Software Implementation

- High-speed — fast AES and binary field multiplication
- Secure — timing-resistant
  - No loop bounds, branches nor table lookups depending on secret data
- Explore the use of hardware support (AES and binary polynomial multiplication)
- Field multiplication in $\mathbb{F}_{2^{128}}$
  - Binary polynomial multiplication (128 $\times$ 128-bit $\rightarrow$ 256-bit)
  - Reduction modulo $f(z) = z^{128} + z^7 + z^2 + z + 1$ (256-bit $\rightarrow$ 128-bit)

Binary Polynomial Multiplication

- Old approach, without hardware support: precomputed tables (López-Dahab multiplication)
- ARMv7 (Câmara, Gouvêa, López 2013)
  - `VMULL.P8`
  - 64×64-bit multiplier using eight `VMULL.P8` invocations
  - 128×128-bit multiplier using three invocations (Karatsuba)

Binary Polynomial Multiplication: ARMv8 AArch32

```
1: vmull.p64 r0q, al, bl
2: vmull.p64 r1q, ah, bh
3: veor th, bl, bh
4: veor tl, al, ah
5: vmull.p64 tq, th, tl
6: veor tq, r0q
7: veor tq, r1q
8: veor r0h, tl
9: veor r1l, th
```

- 64×64-bit multiplier: `VMULL.P64`
- 128×128-bit multiplier using three invocations (Karatsuba)

Binary Polynomial Multiplication: ARMv8 AArch64

```
1: pmull r0.1q, a.1d, b.1d
2: pmull2 r1.1q, a.2d, b.2d
3: ext.16b t0, b, b, #8
4: pmull t1.1q, a.1d, t0.1d
5: pmull2 t0.1q, a.2d, t0.2d
6: eor.16b t0, t0, t1
7: ext.16b t1, z, t0, #8
8: eor.16b r0, r0, t1
9: ext.16b t1, t0, z, #8
10: eor.16b r1, r1, t1
```

- 64×64-bit multiplier: `PMULL`, `PMULL2`
- 128×128-bit multiplier using four invocations
- Karatsuba not used since addressing the upper 64 bits is not directly supported

GCM Bit Reflection

- ARMv7, ARMv8 AArch32
  - No direct support for reversing bits of each byte
  - We use the reflected reduction trick (Gueron and Kounavis 2010)
  - Inversion of bytes in 16-byte vector: `VREV64.8`, `VSWP`
- ARMv8 AArch64
  - `RBIT` reverses bits of each byte in byte vector

Modular Reduction

- Classic approach: shift and xors (Polyakov 2014)
- Multiplier approach: reduction by $f(z) = z^{128} + r(z)$ can be carried out with multiplication by $r(z)$
- ARMv7
  - `VMULL.P8` awkward to use, worse performance
- ARMv8
  - `VMULL.P64`, `PMULL`
- Lazy reduction (Gueron 2010)
  - $Y_i = [(X_i \cdot H) \oplus (X_{i-1} \cdot H^2) \oplus (X_{i-2} \cdot H^3) \oplus (X_{i-3} \cdot H^4)] \bmod f(z)$

AES

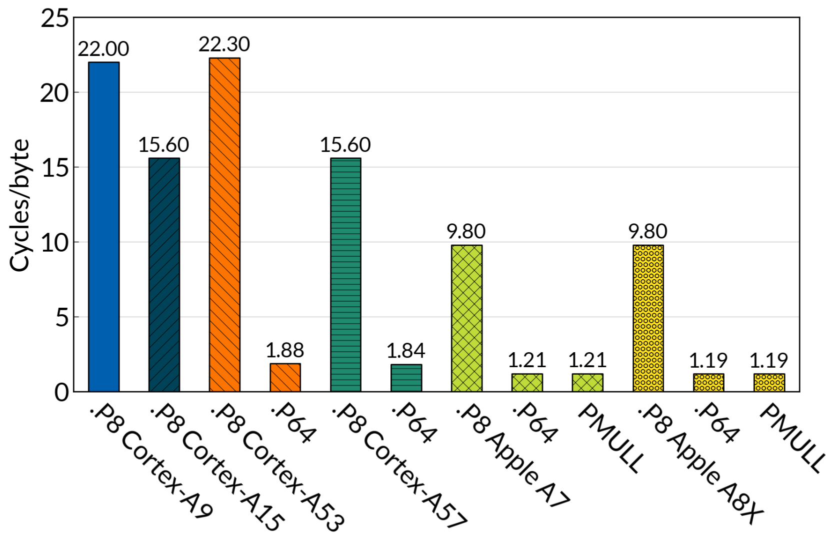
- ARMv7
  - No instruction support
  - Timing-resistant bitsliced implementation from Bernstein and Schwabe
- ARMv8
  - Instruction support
  - Two-block interleaving to avoid hazards
  - Expanded AES key entirely kept in NEON registers
  - Key schedule requires S-box lookups
    - `AESE` can be used (reverting ShiftRows, zero AddRoundKey)

Benchmark

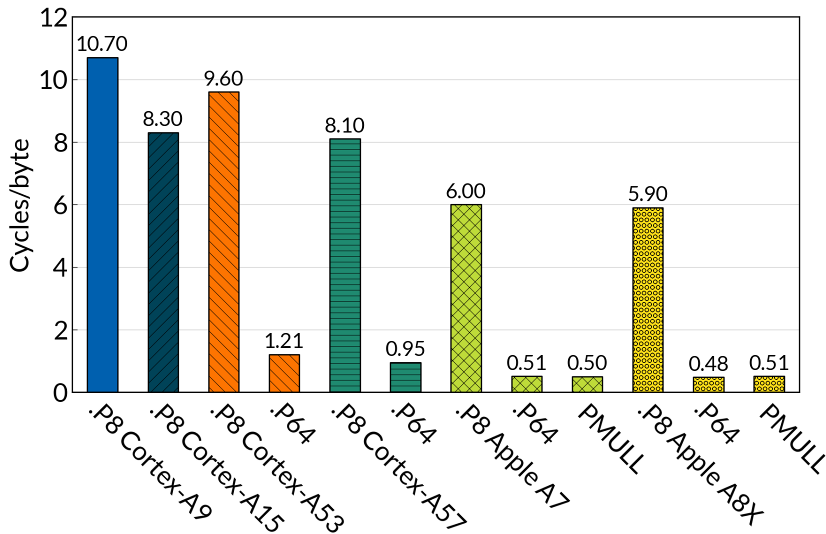
- 10000-byte message inside loop with 256 iterations
- SUPERCOP not yet used — no support for iOS and Android
- Three implementations: `.P8` (with bitsliced AES), `.P64` and `PMULL`
- Five devices:

Device	Architecture	Core	GHz
PandaBoard	ARMv7	Cortex-A9	1.0
Arndale	ARMv7	Cortex-A15	1.7
Galaxy Note 4	ARMv8 AArch32	Cortex-A53/A57	1.3/1.9
iPhone 5s	ARMv8	Apple A7	1.3
iPad Air 2	ARMv8	Apple A8X	1.5

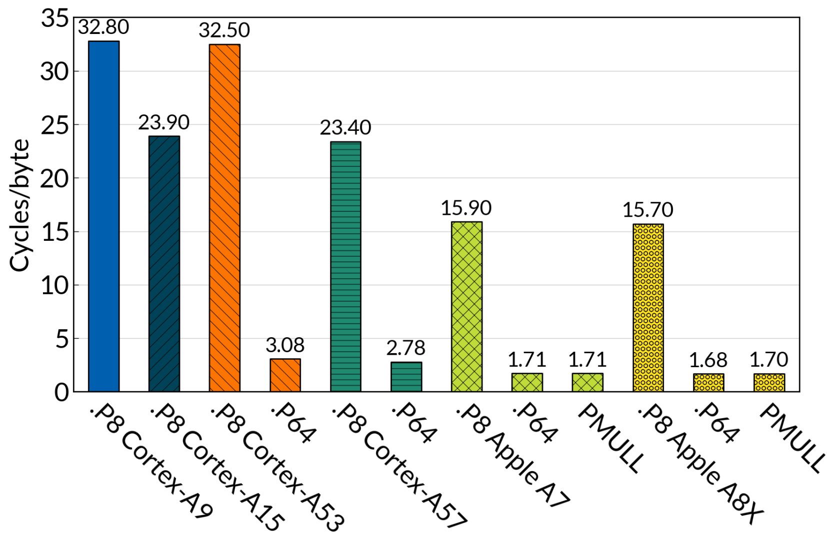
AES-CTR Performance



GHASH Performance



GCM Performance



Conclusion

- Efficient and secure GCM implementation for ARM devices
- New ARMv8 64-bit binary polynomial multiplier coupled with AES instructions: 8–10 times faster
- Natural timing-resistance, no branches nor table lookups required over secret data
- Future work on ARMv8: extend to larger binary fields, apply to elliptic curve cryptography
- Code available, MIT License:

https://github.com/conradoplg/authenc

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 21 - 24, 2015

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 Analysis
- Implementation of Secure Field Multiplication

Results and Comparison

Conclusion

Outline

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

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

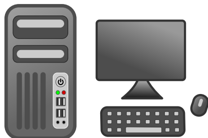
Implementation
- ARM NEON
- Performance Analysis
- Implementation of Secure Field Multiplication

Results and Comparison

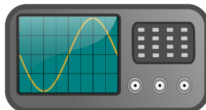
Conclusion

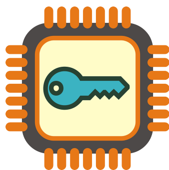
Introduction

Workstation



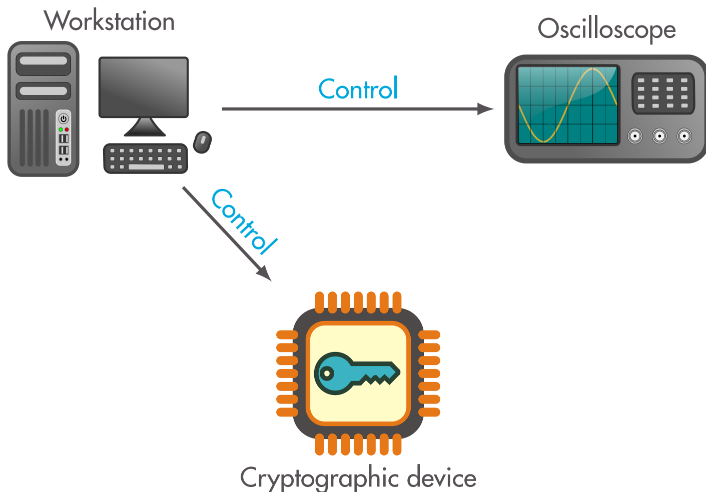
Oscilloscope



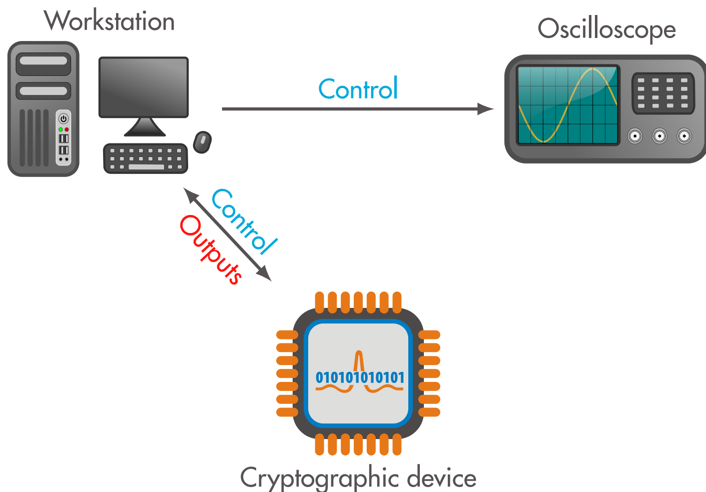


Cryptographic device

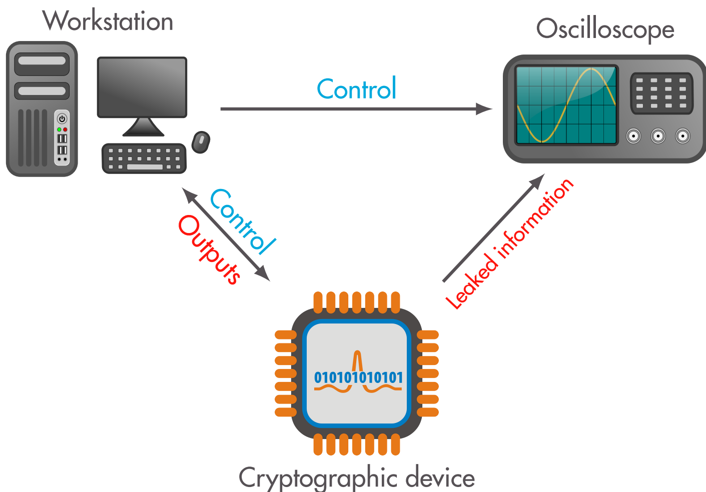
Introduction



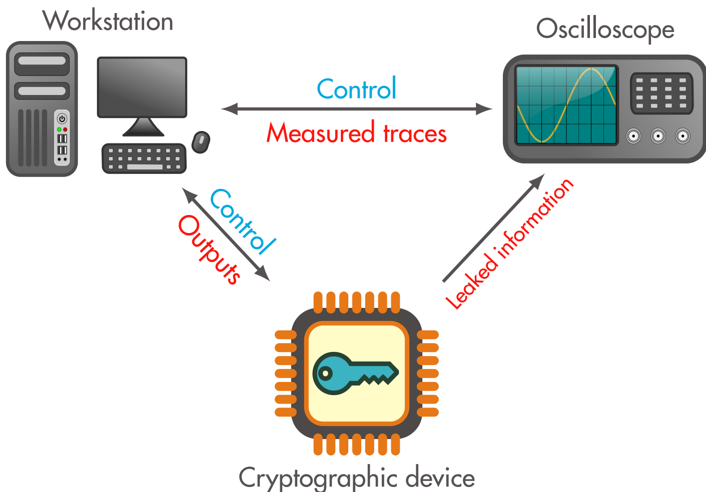
Introduction



Introduction



Introduction



Introduction

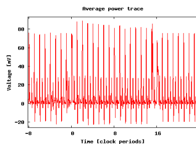
Differential Power Analysis (DPA) [KJJ99]

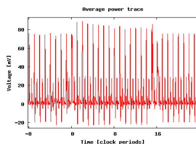
Group by some known or predicted data

000

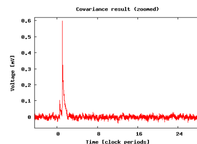
111

Average trace





Differential trace



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

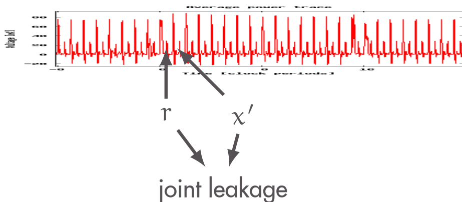
- Suppose x is a sensitive intermediate variable in a block cipher.
- Generate a random r, and process r and masked value

$$x' = x \oplus r$$

  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.

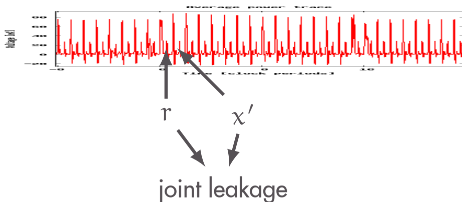
Introduction

- 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

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

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

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

Results and Comparison

Conclusion

Background

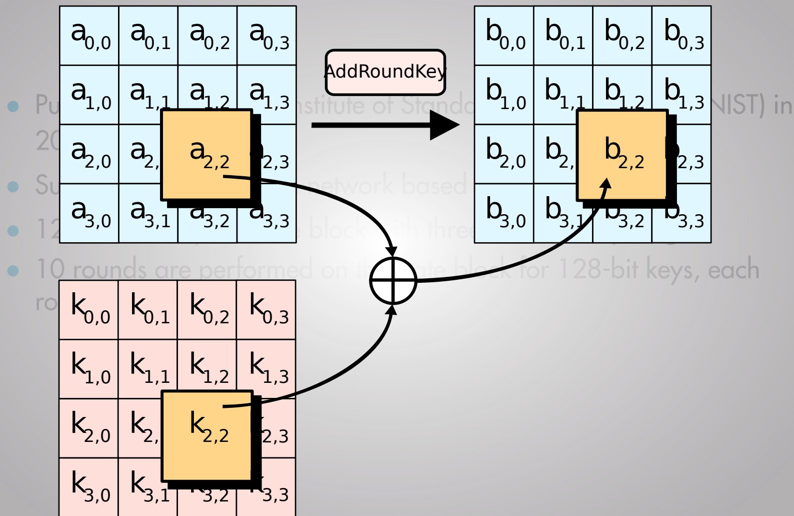
- Published by National Institute of Standards and Technology (NIST) in 2001
- Substitution-permutation network based block cipher
- 128-bit (4*4 bytes) state block with three different key lengths

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

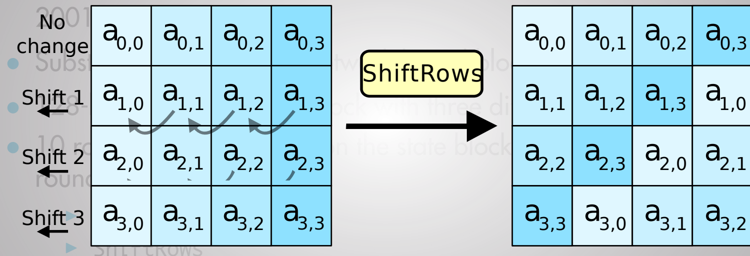


$a_{0,0}$ $a_{0,1}$ $a_{0,2}$ $a_{0,3}$
$a_{1,0}$ $a_{1,1}$ $a_{1,2}$ $a_{1,3}$
$a_{2,0}$ $a_{2,2}$ $a_{2,3}$
$a_{3,0}$ $a_{3,1}$ $a_{3,2}$ $a_{3,3}$
AddRoundKey
$b_{0,0}$ $b_{0,1}$ $b_{0,2}$ $b_{0,3}$
$b_{1,0}$ $b_{1,1}$ $b_{1,2}$ $b_{1,3}$
$b_{2,0}$ $b_{2,2}$ $b_{2,3}$
$b_{3,0}$ $b_{3,1}$ $b_{3,2}$ $b_{3,3}$
$k_{0,0}$ $k_{0,1}$ $k_{0,2}$ $k_{0,3}$
$k_{1,0}$ $k_{1,1}$ $k_{1,2}$ $k_{1,3}$
$k_{2,0}$ $k_{2,2}$ $k_{2,3}$
$k_{3,0}$ $k_{3,1}$ $k_{3,2}$ $k_{3,3}$

Background

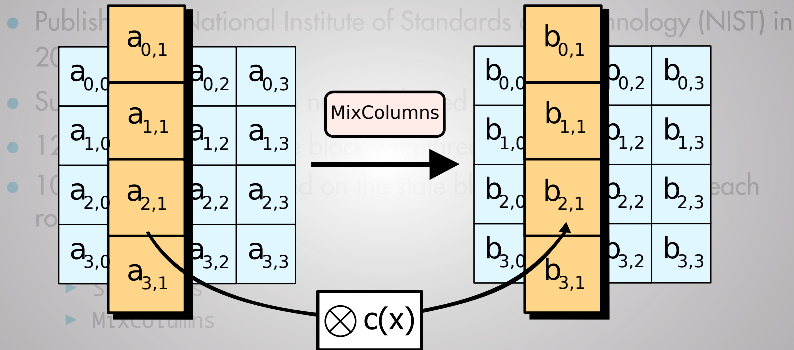
- Published by National Institute of Standards and Technology (NIST) in 2001
- Substitution-permutation network based block cipher
- 128-bit (4*4 bytes) state block with three different key lengths
- 10 rounds are performed on the state block for 128-bit keys, each round contains:
  - `AddRoundKey`
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

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



Background

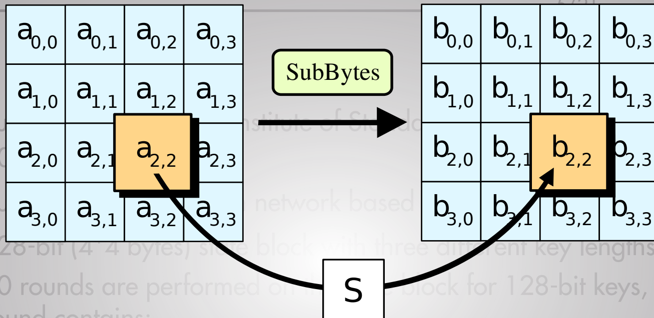
- Published by National Institute of Standards and Technology (NIST) in 2001
- Substitution-permutation network based block cipher
- 128-bit (4*4 bytes) state block with three different key lengths
- 10 rounds are performed on the state block for 128-bit keys, each round contains:
    - `AddRoundKey`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



$a_{0,1}$
$a_{0,0}$
$a_{0,2}$
$a_{0,3}$
$a_{1,0}$
$a_{1,1}$
$a_{1,2}$
$a_{1,3}$
$a_{2,0}$
$a_{2,1}$
$a_{2,2}$
$a_{2,3}$
$a_{3,0}$
$a_{3,1}$
$a_{3,2}$
$a_{3,3}$
MixColumns
$\otimes$ c(x)
$b_{0,0}$
$b_{0,1}$
$b_{0,2}$
$b_{0,3}$
$b_{1,0}$
$b_{1,1}$
$b_{1,2}$
$b_{1,3}$
$b_{2,0}$
$b_{2,1}$
$b_{2,2}$
$b_{2,3}$
$b_{3,0}$
$b_{3,1}$
$b_{3,2}$
$b_{3,3}$

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



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!!!

Background

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

Background

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$((z_i)_i, (x_i)_i)$ $\triangleright \bigoplus_i y_i = x^3$

Background

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

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: $(z_i)_i \leftarrow (x_i^2)_i$ $\triangleright \bigoplus_i z_i = x^2$
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$((y_i)_i, (w_i)_i)$ $\triangleright \bigoplus_i y_i = x^{252}$

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

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$((y_i)_i, (z_i)_i)$

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$((y_i)_i, (z_i)_i)$

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. 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$((y_i)_i, (z_i)_i)$

- 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 $g(x) = x^{2^k}$.

Background

A Flaw in RP Countermeasure (FSE13)

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 $g(x) = x^{2^k}$.

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 $g(x) = x^{2^k}$.
- 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 $g(x) = x^{2^k}$.
- 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. ~~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 $g(x) = x^{2^k}$.
- 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 Analysis
  - Implementation of Secure Field Multiplication
- Results and Comparison
- Conclusion

Implementation

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

Implementation

- ARM is a family of embedded processors
  - Low-cost, high-performance and energy-efficient
  - 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

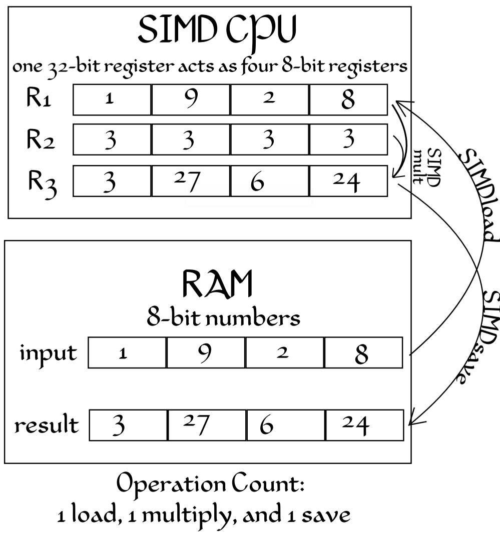


Figure: SIMD Example

Implementation

- ARM is a family of embedded processors
  - Low-cost, high-performance and energy-efficient
  - 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-efficient
  - 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-efficient
  - 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-efficient
  - 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-efficient
  - 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`

Implementation

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

Implementation

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

Multiplication with Barrett modular reduction:

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

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

```
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
```

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

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 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 required by vector implementation of each function, where n is the number of shares

Comparison

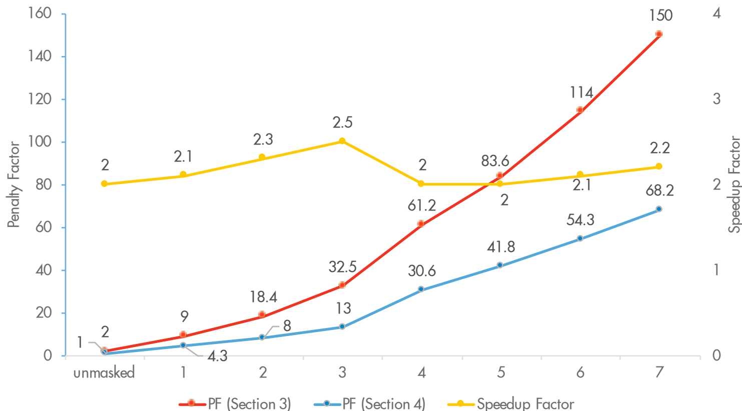


Figure: Penalty factor (PF) of our implementation ([RP10]) in Section 3 and improved implementation (based on [KHL11]) in Section 4; Speedup factor of improved implementation in Section 4 compared to implementation in Section 3.

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

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

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

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

Results and Comparison

Conclusion

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 implementations 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 implementations 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 $\mathrm{GF}(2^8) \triangleq \mathrm{GF}((2^4)^2)$.
- Our implementations achieve a not bad penalty factor, hence, they are deployable in practice.

Question?