Fast symmetric crypto on embedded CPUs

Peter Schwabe
Radboud University Nijmegen, The Netherlands

June 5, 2014

Summer School on the design and security of cryptographic algorithms and devices for real-world applications
Embedded CPUs

4-bit CPUs
- TMS 1000
- Intel 4004
- Atmel MARC4
- Toshiba TLCS-47

8-bit CPUs
- Atmel AVR
- Intel 8051
- Microchip Technology PIC
- STMicroelectronics STM8

16-bit CPUs
- TI MSP430
- Microchip Technology PIC24

32-bit CPUs
- ARM11
- ARM Cortex-M*
- ARM Cortex-A*
- Atmel AVR32
- MIPS32
- AIM 32-bit PowerPC
- STMicroelectronics STM32

Fast symmetric crypto on embedded CPUs
Symmetric crypto
Symmetric crypto

AES
Symmetric crypto

- Salsa
- Rabbit
- Grain
- MICKEY
- HC
- SOSEMANUK
- Trivium

Other algorithms:
- CryptMT
- F-FCSR-H
- Hermes
- MAG
- ABC
- LEX
- NLS
- TSC
- Phelix
- SSS
- DECIM
- VEST
- Mir
- SFINKS
- POMARANCH
- Frogbit
- Yamb
- MOUSTIQUE
- Edon
- TRBDK
- Achterbahn
- WG
- ZK-Crypt
- Polar
- YAEA

Fast symmetric crypto on embedded CPUs
Symmetric crypto
Symmetric crypto

Minalpher SHELL STRIBOB
POLAWIS Deoxys ICEPOLE $\pi$-Cipher
OMD Julius AES-OTR Tiaoxin
Wheesht ELmd AES-CPFB LAC
Hsignal-SIV AES-CMCC CBA Prøst
Keyak Ascon AEGIS Calico
MORUS Joltik AE AEZ PAEQ
Ketje ACORN Artemia SILC
OCl AES-COPA CLOC Silver
SCREAM AES-JAMB iFeed[AES]
Enchilada KIASU TriviA-ck
Raviyoyla YAES POET Sablier Marble
Optimizing crypto

- This talk: optimize for speed
- Implement algorithms in assembly
- Available instructions and registers are determined by the target architecture
Optimizing crypto

- This talk: optimize for speed
- Implement algorithms in assembly
- Available instructions and registers are determined by the target **architecture**
- **Throughput**: number of instructions (of a certain type) we can do per cycle
Optimizing crypto

- This talk: optimize for speed
- Implement algorithms in assembly
- Available instructions and registers are determined by the target architecture
- **Throughput**: number of instructions (of a certain type) we can do per cycle
- **Latency** of an instruction: number of cycles we have to wait before using the result
Optimizing crypto

- This talk: optimize for speed
- Implement algorithms in assembly
- Available instructions and registers are determined by the target architecture
- **Throughput**: number of instructions (of a certain type) we can do per cycle
- **Latency** of an instruction: number of cycles we have to wait before using the result
- Latency and throughput are determined by the microarchitecture
Optimizing crypto

- This talk: optimize for speed
- Implement algorithms in assembly
- Available instructions and registers are determined by the target architecture
- **Throughput**: number of instructions (of a certain type) we can do per cycle
- **Latency** of an instruction: number of cycles we have to wait before using the result
- Latency and throughput are determined by the microarchitecture
- Optimizing software in assembly means:
  - Find good representation of data
  - Choose suitable instructions that implement the algorithm
  - Schedule those instruction to hide latencies
  - Assign registers efficiently (avoid spills)
Keccak on ARM11

Joint work with Bo-Yin Yang and Shang-Yi Yang
The ARM11

- 16 32-bit integer registers (1 used as PC, one used as SP): 14 freely available
- Executes at most one instruction per cycle
- 1 cycle latency for all relevant arithmetic instructions, 3 cycles for loads from cache
- Standard 32-bit RISC instruction set; two exceptions:
The ARM11

- 16 32-bit integer registers (1 used as PC, one used as SP): 14 freely available
- Executes at most one instruction per cycle
- 1 cycle latency for all relevant arithmetic instructions, 3 cycles for loads from cache
- Standard 32-bit RISC instruction set; two exceptions:
  - One input of arithmetic instructions can be rotated or shifted for free as part of the instruction
  - This input is needed one cycle earlier in the pipeline \( \Rightarrow \) “backwards latency” + 1

Fast symmetric crypto on embedded CPUs
The ARM11

- 16 32-bit integer registers (1 used as PC, one used as SP): 14 freely available
- Executes at most one instruction per cycle
- 1 cycle latency for all relevant arithmetic instructions, 3 cycles for loads from cache
- Standard 32-bit RISC instruction set; two exceptions:
  - One input of arithmetic instructions can be rotated or shifted for free as part of the instruction
  - This input is needed one cycle earlier in the pipeline ⇒ “backwards latency” + 1
  - Loads and stores can move 64-bits between memory and 2 adjacent 32-bit registers (same cost as 32-bit load/store)
Keccak

- State of $5 \times 5$ matrix of 64-bit lanes
- Absorb message in blocks of 128 bytes
- Perform state transformation in 24 rounds; each round:
  - Compute $b_0, \ldots, b_4$ as XORs of columns
  - Compute $c_0, \ldots, c_4$, each as $b_i \oplus (b_j \ll 1)$
Keccak

- State of $5 \times 5$ matrix of 64-bit lanes
- Absorb message in blocks of 128 bytes
- Perform state transformation in 24 rounds; each round:
  - Compute $b_0, \ldots, b_4$ as XORs of columns
  - Compute $c_0, \ldots, c_4$, each as $b_i \oplus (b_j \ll 1)$
  - Update state columnwise
  - Pick up 5 lanes from a diagonal
  - XOR each lane with one of the $c_i$
  - Rotate each lane by a different fixed distance
  - Obtain each new lanes as $l_i \oplus ((\neg l_j) \& l_k)$
Keccak

- State of $5 \times 5$ matrix of 64-bit lanes
- Absorb message in blocks of 128 bytes
- Perform state transformation in 24 rounds; each round:
  - Compute $b_0, \ldots, b_4$ as XORs of columns
  - Compute $c_0, \ldots, c_4$, each as $b_i \oplus (b_j \ll 1)$
  - Update state columnwise
  - Pick up 5 lanes from a diagonal
  - XOR each lane with one of the $c_i$
  - Rotate each lane by a different fixed distance
  - Obtain each new lanes as $l_i \oplus ((\neg l_j) \& l_k)$
  - One lane per column is additionally XORed with a round constant
A 64-bit hash-function on a 32-bit CPU

- Represent each lane in two registers, XOR and AND are trivial
- How about 64-bit rotate with 32-bit registers?
A 64-bit hash-function on a 32-bit CPU

- Represent each lane in two registers, XOR and AND are trivial
- How about 64-bit rotate with 32-bit registers?
- Answer by the Keccak implementation guide: bit interleaving
- Put all bits from even positions into one 32-bit register, all odd bits into the other
- Perform all rotates for free on 32-bit registers
A 64-bit hash-function on a 32-bit CPU

- Represent each lane in two registers, XOR and AND are trivial
- How about 64-bit rotate with 32-bit registers?
- Answer by the Keccak implementation guide: *bit interleaving*
- Put all bits from even positions into one 32-bit register, all odd bits into the other
- Perform all rotates for free on 32-bit registers
- $a \leftarrow b \odot (c \ll n)$ is free rotation, but $a \leftarrow (b \odot c) \ll n$ is not
A 64-bit hash-function on a 32-bit CPU

- Represent each lane in two registers, XOR and AND are trivial
- How about 64-bit rotate with 32-bit registers?
- Answer by the Keccak implementation guide: bit interleaving
- Put all bits from even positions into one 32-bit register, all odd bits into the other
- Perform all rotates for free on 32-bit registers
- \( a \leftarrow b \odot (c \ll n) \) is free rotation, but \( a \leftarrow (b \odot c) \ll n \) is not
- Don’t rotate output, rotate for free when the value is used as input
- When both inputs of an instruction need to be rotated:
  \[
  a \leftarrow (b \ll n_1) \odot (c \ll n_2).
  \]
- Compute:
  \[
  a \leftarrow b \odot (c \ll (n_2 - n_1))
  \]
  and set the implicit rotation distance of \( a \) to \( n_1 \)
A 64-bit hash-function on a 32-bit CPU

- Represent each lane in two registers, XOR and AND are trivial
- How about 64-bit rotate with 32-bit registers?
- Answer by the Keccak implementation guide: *bit interleaving*
- Put all bits from even positions into one 32-bit register, all odd bits into the other
- Perform all rotates for free on 32-bit registers
- \( a \leftarrow b \odot (c \ll n) \) is free rotation, but \( a \leftarrow (b \odot c) \ll n \) is not
- Don’t rotate output, rotate for free when the value is used as input
- When both inputs of an instruction need to be rotated:
  \[
  a \leftarrow (b \ll n_1) \odot (c \ll n_2).
  \]
- Compute:
  \[
  a \leftarrow b \odot (c \ll (n_2 - n_1))
  \]
  and set the implicit rotation distance of \( a \) to \( n_1 \)
- Need to keep implicit rotation distances invariant over loop iterations
- Full unrolling essentially makes all rotates free
Memory access overhead

- 200-byte state is way too large for 56 register bytes
- Simple structure of main transformations:
  - Load 5 half-lanes
  - Load 5 values $c_i$
  - Perform arithmetic (10 XOR, 5 AND)
  - Store 5 result lanes

This means 50% load/store overhead

Even worse for computation of $b_i$ and $c_i$

Not easy to use 64-bit loads and stores (needs smart memory layout)

Can eliminate some loads of $c_i$, but still huge overhead

Overall we have 4800 arithmetic instructions in 24 rounds

Lower bound on performance: $4800 / 128 = 37.5$ cycles/byte

Actual performance: 79.32 cycles/byte

Fast symmetric crypto on embedded CPUs
Memory access overhead

- 200-byte state is way too large for 56 register bytes
- Simple structure of main transformations:
  - Load 5 half-lanes
  - Load 5 values $c_i$
  - Perform arithmetic (10 XOR, 5 AND)
  - Store 5 result lanes
- This means 50% load/store overhead
- Even worse for computation of $b_i$ and $c_i$
Memory access overhead

- 200-byte state is way too large for 56 register bytes
- Simple structure of main transformations:
  - Load 5 half-lanes
  - Load 5 values $c_i$
  - Perform arithmetic (10 XOR, 5 AND)
  - Store 5 result lanes
- This means 50% load/store overhead
- Even worse for computation of $b_i$ and $c_i$
- Not easy to use 64-bit loads ands stores (needs smart memory layout)
- Can eliminate some loads of $c_i$, but still huge overhead
Memory access overhead

- 200-byte state is way too large for 56 register bytes
- Simple structure of main transformations:
  - Load 5 half-lanes
  - Load 5 values $c_i$
  - Perform arithmetic (10 XOR, 5 AND)
  - Store 5 result lanes
- This means 50% load/store overhead
- Even worse for computation of $b_i$ and $c_i$
- Not easy to use 64-bit loads ands stores (needs smart memory layout)
- Can eliminate some loads of $c_i$, but still huge overhead
- Overall we have 4800 arithmetic instructions in 24 rounds
- Lower bound on performance: $4800/128 = 37.5$ cycles/byte
Memory access overhead

- 200-byte state is way too large for 56 register bytes
- Simple structure of main transformations:
  - Load 5 half-lanes
  - Load 5 values $c_i$
  - Perform arithmetic (10 XOR, 5 AND)
  - Store 5 result lanes
- This means 50% load/store overhead
- Even worse for computation of $b_i$ and $c_i$
- Not easy to use 64-bit loads ands stores (needs smart memory layout)
- Can eliminate some loads of $c_i$, but still huge overhead
- Overall we have 4800 arithmetic instructions in 24 rounds
- Lower bound on performance: $4800/128 = 37.5$ cycles/byte
- Actual performance: 79.32 cycles/byte
Salsa20 on ARM Cortex-A8

Joint work with Daniel J. Bernstein
The ARM Cortex-A8

The ARM core

- Essentially the same instruction set as ARM 11
- Again, 16 integer registers, 14 freely available
- Can issue two instructions per cycle
- Only one load/store per cycle
- More serious latency constraints than ARM11
The ARM Cortex-A8

The ARM core

- Essentially the same instruction set as ARM 11
- Again, 16 integer registers, 14 freely available
- Can issue two instructions per cycle
- Only one load/store per cycle
- More serious latency constraints than ARM11

The NEON vector unit

- 16 128-bit vector registers
- One arithmetic + one load/store/shuffle per cycle
- No free shifts or rotates
- Fairly complex latency rules
Salsa20

- Generates random stream in 64-byte blocks, works on 32-bit integers
- Blocks are independent
- Per block: 20 rounds; each round doing 16 add-rotate-xor sequences, such as
  \[
  s_4 = x_0 + x_{12} \\
  x_4 \leftarrow (s_4 \ggg 25)
  \]
- These sequences are 4-way parallel
Salsa20

- Generates random stream in 64-byte blocks, works on 32-bit integers
- Blocks are independent
- Per block: 20 rounds; each round doing 16 add-rotate-xor sequences, such as
  \[ s_4 = x_0 + x_{12} \]
  \[ x_4 ^= (s_4 >>> 25) \]
- These sequences are 4-way parallel
- In ARM without NEON: 2 instructions, 1 cycle
- Sounds like total of \((20 \cdot 16)/64 = 5\) cycles/byte
Salsa20

- Generates random stream in 64-byte blocks, works on 32-bit integers
- Blocks are independent
- Per block: 20 rounds; each round doing 16 add-rotate-xor sequences, such as
  \[ s_4 = x_0 + x_{12} \]
  \[ x_4 \leftarrow (s_4 \gg 25) \]
- These sequences are 4-way parallel
- In ARM without NEON: 2 instructions, 1 cycle
- Sounds like total of \((20 \cdot 16)/64 = 5\) cycles/byte, but:
  - Only 14 integer registers (need at least 17)
  - Latencies cause big trouble
  - Actual implementations slower than 15 cycles/byte
A first approach in NEON

- Per round do $4 \times$ something like:
  
  $4x \ a0 = \text{diag1 } + \text{diag0}$
  
  $4x \ b0 = a0 \ll 7$
  
  $4x \ a0 \text{ unsigned } \gg 25$
  
  $\text{diag3 } \sim b0$
  
  $\text{diag3 } \sim a0$

- + some (free) shuffles

Intuitive cycle lower bound:

$$\frac{5 \times 4 \times 20}{64} = 6.25 \text{ cycles/byte}$$

Problem: The above sequence has a $9$-cycle latency, thus:

$$\frac{9 \times 4 \times 20}{64} = 11.25 \text{ cycles/byte}$$
A first approach in NEON

- Per round do $4 \times$ something like:
  - $4x \ a0 = \text{diag1} + \text{diag0}$
  - $4x \ b0 = a0 \ll 7$
  - $4x \ a0$ unsigned $\gg= 25$
    - $\text{diag3} \ = \ b0$
    - $\text{diag3} \ = \ a0$
- + some (free) shuffles
- Intuitive cycle lower bound:
  - $(5 \cdot 4 \cdot 20)/64 = 6.25$ cycles/byte
A first approach in NEON

- Per round do $4 \times$ something like:
  
  \begin{align*}
  4x \ a0 &= \text{diag}1 + \text{diag}0 \\
  4x \ b0 &= a0 \ll 7 \\
  4x \ a0 \text{ unsigned} &\gg= 25 \\
  \text{diag}3 &= b0 \\
  \text{diag}3 &= a0
  \end{align*}

- + some (free) shuffles

- Intuitive cycle lower bound:
  \[(5 \cdot 4 \cdot 20)/64 = 6.25 \text{ cycles/byte}\]

- Problem: The above sequence has a 9-cycle latency, thus:
  \[(9 \cdot 4 \cdot 20)/64 = 11.25 \text{ cycles/byte}\]
Trading parallelism

- Salsa20 rounds have 4-way data-level parallelism
- In a scalar implementations this turns into 4-way instruction-level parallelism
Trading parallelism

- Salsa20 rounds have 4-way data-level parallelism
- In a scalar implementations this turns into 4-way instruction-level parallelism
- Good for pipelined and superscalar execution

Lower bound when interleaving 2 blocks: 6.875 cycles/byte
Lower bound when interleaving 3 blocks: 6.25 cycles/byte
Trading parallelism

- Salsa20 rounds have 4-way data-level parallelism
- In a scalar implementations this turns into 4-way instruction-level parallelism
- Good for pipelined and superscalar execution
- The vector implementation needs 4-way data parallelism, there is (almost) no instruction-level parallelism left
- Bad for pipelined and superscalar execution
Salsa20 rounds have 4-way data-level parallelism

In a scalar implementations this turns into 4-way instruction-level parallelism

Good for pipelined and superscalar execution

The vector implementation needs 4-way data parallelism, there is (almost) no instruction-level parallelism left

Bad for pipelined and superscalar execution

Idea: Blocks are independent, use this to re-introduce instruction-level parallelism
Trading parallelism

- Salsa20 rounds have 4-way data-level parallelism.
- In a scalar implementation, this turns into 4-way instruction-level parallelism.
- Good for pipelined and superscalar execution.
- The vector implementation needs 4-way data parallelism, there is (almost) no instruction-level parallelism left.
- Bad for pipelined and superscalar execution.
- Idea: Blocks are independent, use this to re-introduce instruction-level parallelism.
- Lower bound when interleaving 3 blocks: 6.25 cycles/byte.
Going even further

- NEON is basically a coprocessor to the ARM core
- ARM decodes instructions, forwards NEON instructions to the NEON unit
Going even further

- NEON is basically a coprocessor to the ARM core
- ARM decodes instructions, forwards NEON instructions to the NEON unit
- Idea: Also keep the ARM core busy with Salsa20 computations
- New bottleneck: ARM core decodes at most 2 instructions per cycle
Going even further

- NEON is basically a coprocessor to the ARM core
- ARM decodes instructions, forwards NEON instructions to the NEON unit
- Idea: Also keep the ARM core busy with Salsa20 computations
- New bottleneck: ARM core decodes at most 2 instructions per cycle
- Add-rotate-xor is only 2 ARM instructions
- Best tradeoff: One block on ARM, two blocks on NEON
A flavor of the code

```
4x a0 = diag1 + diag0
    4x next_a0 = next_diag1 + next_diag0
    s4 = x0 + x12
    s9 = x5 + x1

4x b0 = a0 << 7
    4x next_b0 = next_a0 << 7

4x a0 unsigned >>= 25
    4x next_a0 unsigned >>= 25
    x4 ^= (s4 >>> 25)
    x9 ^= (s9 >>> 25)
    s8 = x4 + x0
    s13 = x9 + x5

    diag3 ^= b0
        next_diag3 ^= next_b0
    diag3 ^= a0
        next_diag3 ^= next_a0
    x8 ^= (s8 >>> 23)
    x13 ^= (s13 >>> 23)
```
5.47 cycles/byte for Salsa20 encryption on ARM Cortex-A8 with NEON
The case of AES
Importance of AES

- Most widely used symmetric crypto algorithm
- Used in many constructions:
  - 10 SHA-3 submissions were AES-based
  - 25 CAESAR submissions use AES
- Only accepted encryption algorithm for various security certifications
- You need a stream cipher? “Use AES-CTR”
AES on 32-bit processors

- Idea from the AES proposal: Merge SubBytes, ShiftRows, and MixColumns
- Use 4 lookup tables T0, T1, T2, and T3 (1 KB each)
AES on 32-bit processors

- Idea from the AES proposal: Merge SubBytes, ShiftRows, and MixColumns
- Use 4 lookup tables T0, T1, T2, and T3 (1 KB each)

The first round of AES in C

- Input: 32-bit integers y0, y1, y2, y3
- Output: 32-bit integers z0, z1, z2, z3
- Round keys in 32-bit-integer array rk[44]

```c
z0 = T0[ y0 >> 24 ] ^ T1[(y1 >> 16) & 0xff] ^ T2[(y2 >> 8) & 0xff] ^ T3[ y3 & 0xff ] ^ rk[4];
z1 = T0[ y1 >> 24 ] ^ T1[(y2 >> 16) & 0xff] ^ T2[(y3 >> 8) & 0xff] ^ T3[ y0 & 0xff ] ^ rk[5];
z2 = T0[ y2 >> 24 ] ^ T1[(y3 >> 16) & 0xff] ^ T2[(y0 >> 8) & 0xff] ^ T3[ y1 & 0xff ] ^ rk[6];
z3 = T0[ y3 >> 24 ] ^ T1[(y0 >> 16) & 0xff] ^ T2[(y1 >> 8) & 0xff] ^ T3[ y2 & 0xff ] ^ rk[7];
```
Foot-shooting prevention

Foot-Shooting Prevention Agreement

I, ________, promise that once
Your Name
I see how simple AES really is, I will
not implement it in production code
even though it would be really fun.

This agreement shall be in effect
until the undersigned creates a
meaningful interpretive dance that
compares and contrasts cache-based,
timing, and other side channel attacks
and their countermeasures.

Signature  Date

The problem with $T$ tables

- $T$ tables perform loads from secret locations
- Timing information leaks memory addresses

Fast symmetric crypto on embedded CPUs
The problem with $T$ tables

- $T$ tables perform loads from secret locations
- Timing information leaks memory addresses
- Easiest case: Cache timing
  - Load of data in cache is fast
  - Load of data not in cache is slow

Various other sources for timing leaks from memory access


To put it bluntly:

- AES is a well understood secure algorithm
- Implementations of AES are horribly insecure

Fast symmetric crypto on embedded CPUs
The problem with $T$ tables

- $T$ tables perform loads from secret locations
- Timing information leaks memory addresses
- Easiest case: Cache timing
  - Load of data in cache is fast
  - Load of data not in cache is slow
- Various other sources for timing leaks from memory access
The problem with $T$ tables

- $T$ tables perform loads from secret locations
- Timing information leaks memory addresses
- Easiest case: Cache timing
  - Load of data in cache is fast
  - Load of data not in cache is slow
- Various other sources for timing leaks from memory access
The problem with $T$ tables

- $T$ tables perform loads from secret locations
- Timing information leaks memory addresses
- Easiest case: Cache timing
  - Load of data in cache is fast
  - Load of data not in cache is slow
- Various other sources for timing leaks from memory access
- To put it bluntly:
  - AES is a well understood secure algorithm
  - Implementations of AES are horribly insecure
How could AES be chosen?

“Table lookup: not vulnerable to timing attacks; relatively easy to effect a defense against power attacks by software balancing of the lookup address.”

Modern AES software

\( T \) tables

- Use only on machines with constant-time loads
- Caches are *not* the only problem
- Use assembly to prevent (?) foot-shooting
Modern AES software

$T$ tables

- Use only on machines with constant-time loads
- Caches are *not* the only problem
- Use assembly to prevent (?) foot-shooting

Bitslicing

- Transpose binary state matrix in registers
- Simulate hardware implementation in software
- Needs fast XOR and AND instructions
- Example: 384 bit operations per cycle on 64-bit Intel CPUs
Modern AES software

$T$ tables

- Use only on machines with constant-time loads
- Caches are *not* the only problem
- Use assembly to prevent (?) foot-shooting

Bitslicing

- Transpose binary state matrix in registers
- Simulate hardware implementation in software
- Needs fast XOR and AND instructions
- Example: 384 bit operations per cycle on 64-bit Intel CPUs

Vector permutes

- Implement AES through $\mathbb{F}_{2^8}$ arithmetic
- Represent $\mathbb{F}_{2^8}$ as quadratic extension of $\mathbb{F}_{2^4}$
- Use vector-permute instructions as lookups
- Needs fast and powerful vector-permute instructions
- Example: AltiVec, NEON(?)
Modern AES software

$T$ tables
- Use only on machines with constant-time loads
- Caches are not the only problem
- Use assembly to prevent (?) foot-shooting

Bitslicing
- Transpose binary state matrix in registers
- Simulate hardware implementation in software
- Needs fast XOR and AND instructions
- Example: 384 bit operations per cycle on 64-bit Intel CPUs

Vector permutes
- Implement AES through $\mathbb{F}_{2^8}$ arithmetic
- Represent $\mathbb{F}_{2^8}$ as quadratic extension of $\mathbb{F}_{2^4}$
- Use vector-permute instructions as lookups
- Needs fast and powerful vector-permute instructions
- Example: AltiVec, NEON (?)

Hardware support
- Intel has AES-NI since Westmere
- ARMv8 has HW AES
Challenges

- Beat our Keccak ARM11 implementation
Challenges

- Beat our Keccak ARM11 implementation
- Implement AES with vector permute in NEON
Challenges

- Beat our Keccak ARM11 implementation
- Implement AES with vector permute in NEON
- Implement AES without $T$ tables in plain ARM
Challenges

SPEED UP

ALL THE CEASAR SUBMISSIONS
References

- SHA-3 finalists on ARM11:
  http://cryptojedi.org/papers/#sha3arm
- NEON crypto:
  http://cryptojedi.org/papers/#neoncrypto
References

▶ SHA-3 finalists on ARM11:
http://cryptojedi.org/papers/#sha3arm

▶ NEON crypto:
http://cryptojedi.org/papers/#neoncrypto

▶ Bitsliced AES:
  www.iacr.org/archive/ches2007/47270121/47270121.ps
  http://cryptojedi.org/papers/#aesbs
References

- SHA-3 finalists on ARM11:
  http://cryptojedi.org/papers/#sha3arm
- NEON crypto:
  http://cryptojedi.org/papers/#neoncrypto
- Bitsliced AES:
    www.iacr.org/archive/ches2007/47270121/47270121.ps
    http://cryptojedi.org/papers/#aesbs