SHA-3 on ARM11 processors

Bo-Yin Yang

中央研究院

Joint work with Peter Schwabe, Shang-Yi Yang

March 22, 2012

3rd SHA-3 Candidate Conference

Introduction

- Most smartphones and tablets and many embedded devices are powered by ARM processors
- One of the most widespread microarchitectures: ARM11 ($>$ 500,000,000 chips sold per year)
- Large portion of those chips is used in environments that want fast crypto

Introduction

Most smartphones and tablets and many embedded devices are powered by ARM processors

One of the most widespread microarchitectures: ARM11 ($>$ 500,000,000 chips sold per year)

Large portion of those chips is used in environments that want fast crypto

Question answered here: How fast are the 256-bit output versions of the 5 remaining SHA-3 candidates on ARM11

Implementations in hand-optmized assembly

Further interpretations of the results:

- Performance of SHA-3 candidates on a “typical” 32-bit microarchitecture
- How good are compilers at optmizing existing C implementations for a simple 32-bit architecture

The ARM11 microarchitecture

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 microarchitecture

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

The ARM11 microarchitecture

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 adjecent 32-bit registers (same cost as 32-bit load/store)

BLAKE

Main work: 14 rounds, each consisting of 8 evaluations of G

Each G: 6 additions, 6 xors, 4 rotations by fixed distances

BLAKE

Main work: 14 rounds, each consisting of 8 evaluations of G

Each G: 6 additions, 6 xors, 4 rotations by fixed distances

Merge rotations of outputs with arithmetic:

- Do not rotate output after instruction, rotate for free when the value is used as input

BLAKE

Main work: 14 rounds, each consisting of 8 evaluations of G

Each G: 6 additions, 6 xors, 4 rotations by fixed distances

Merge rotations of outputs with arithmetic:

- Do not rotate output after instruction, rotate for free when the value is used as input
- Eventually, both inputs of an instruction need to be rotated:

$$a \leftarrow (b \ggg n_1) \odot (c \ggg n_2).$$

- Compute:

$$a \leftarrow b \odot (c \ggg (n_2 - n_1))$$

  and set the implicit rotation distance of a to n_1

BLAKE

Main work: 14 rounds, each consisting of 8 evaluations of G

Each G: 6 additions, 6 xors, 4 rotations by fixed distances

Merge rotations of outputs with arithmetic:

- Do not rotate output after instruction, rotate for free when the value is used as input
- Eventually, both inputs of an instruction need to be rotated:

$$a \leftarrow (b \ggg n_1) \odot (c \ggg n_2).$$

- Compute:

$$a \leftarrow b \odot (c \ggg (n_2 - n_1))$$

  and set the implicit rotation distance of a to n_1
- With full unrolling this eliminates all but the last rotates

BLAKE

Main work: 14 rounds, each consisting of 8 evaluations of G

Each G: 6 additions, 6 xors, 4 rotations by fixed distances

Merge rotations of outputs with arithmetic:

- Do not rotate output after instruction, rotate for free when the value is used as input
- Eventually, both inputs of an instruction need to be rotated:

$$a \leftarrow (b \ggg n_1) \odot (c \ggg n_2).$$

- Compute:

$$a \leftarrow b \odot (c \ggg (n_2 - n_1))$$

  and set the implicit rotation distance of a to n_1
- With full unrolling this eliminates all but the last rotates

Additional optimization: Reduction of loads and stores

Speed: 33.93 cycles/byte for long messages

Grøstl

Main work: 10 rounds, each consisting of permutations P and Q, similar to AES

Use Lookup-table-based approach (similar to AES)

Each round, each permutation: 64 64-bit table lookups and 56 xors of 64-bit values

With suitable tables (8 KB): support 64-bit loads

Use interleaved tables to reduce the size of constant offsets

Speed: 110.16 cycles/byte for long messages

JH

Designed for bitsliced implementations (128-bit or 256-bit vectors)

Main work: 42 rounds, each with 48 logical operations on 128-bit vectors (4 operations on 32-bit words)

JH

Designed for bitsliced implementations (128-bit or 256-bit vectors)

Main work: 42 rounds, each with 48 logical operations on 128-bit vectors (4 operations on 32-bit words)

Full unrolling would result in very large code: unroll 7 loop iterations instead

Two loops: over 4 32-bit words and over 6 blocks of 7 rounds

Reorder loops to avoid frequent loads and stores (requires attention in the last two rounds of each block)

JH

Designed for bitsliced implementations (128-bit or 256-bit vectors)

Main work: 42 rounds, each with 48 logical operations on 128-bit vectors (4 operations on 32-bit words)

Full unrolling would result in very large code: unroll 7 loop iterations instead

Two loops: over 4 32-bit words and over 6 blocks of 7 rounds

Reorder loops to avoid frequent loads and stores (requires attention in the last two rounds of each block)

Additional operation: Swap blocks of adjecent bits (1-bit, 2-bit, 4-bit, . . . 64-bit blocks)

For 16-bit blocks: Use free rotation, for 8-bit blocks use `rev16` instruction

Speed: 156.43 cycles/byte for long messages

Keccak

Keccak is operating on 64-bit words, but no additions involved

Implementation technique suggested by designers for 32-bit architectures: bit interleaving

All bits of odd positions in one 32-bit word, all bits at even positions in another 32-bit word

Advantage: Rotations can be done as 32-bit rotations (free for ARM11)

Keccak

Keccak is operating on 64-bit words, but no additions involved

Implementation technique suggested by designers for 32-bit architectures: bit interleaving

All bits of odd positions in one 32-bit word, all bits at even positions in another 32-bit word

Advantage: Rotations can be done as 32-bit rotations (free for ARM11)

Main work: 24 rounds, each round consists of 150 xors and 50 ands

Merge (almost) all rotations with arithmetic as for Blake

Keccak

Keccak is operating on 64-bit words, but no additions involved

Implementation technique suggested by designers for 32-bit architectures: bit interleaving

All bits of odd positions in one 32-bit word, all bits at even positions in another 32-bit word

Advantage: Rotations can be done as 32-bit rotations (free for ARM11)

Main work: 24 rounds, each round consists of 150 xors and 50 ands

Merge (almost) all rotations with arithmetic as for Blake

Main trouble: Almost 50% overhead from loads and stores

This is *with* use of 64-bit stores

Speed: 71.73 cycles/byte for long messages

Skein

Main work: 72 rounds, each performing 4 MIX operations

Each MIX operation: One 64-bit addition, one 64-bit xor, one 64-bit rotation

After each 4 rounds: “key injection”

Skein

Main work: 72 rounds, each performing 4 MIX operations

Each MIX operation: One 64-bit addition, one 64-bit xor, one 64-bit rotation

After each 4 rounds: “key injection”

Naive implementation has huge overhead from register spills

Optimization consists in rearranging independent MIX operations to reduce number of spills

Skein

Main work: 72 rounds, each performing 4 MIX operations

Each MIX operation: One 64-bit addition, one 64-bit xor, one 64-bit rotation

After each 4 rounds: “key injection”

Naive implementation has huge overhead from register spills

Optimization consists in rearranging independent MIX operations to reduce number of spills

Furthermore, we precompute part of the key injection: speedup by 1.78 cycles/byte

Speed: 42.10 cycles/byte for long messages

Results

Cycles/byte reported by eBASH on a Samsung Galaxy i7500 smart phone (528 MHz ARM11) for long messages (median):

	This paper	Previously fastest in eBASH
Blake	33.93	46.29 (sphlib v3.0)
Grøstl	110.16	140.17 (arm32, assembly!)
JH	156.43	247.16 (bitslice_opt32, round-2 version with only 35.5 rounds)
Keccak	71.73	86.95 (simple32bi)
Skein	42.10	94.57 (sphlib-small v3.0)
SHA-256	26.6	39.19 (sphlib v3.0)

Details for various message lengths and quartiles in the paper.

Results online

All software is in the public domain and included in SUPERCOP

Paper is online at http://cryptojedi.org/papers/#sha3arm