SHA-3 on ARM11 processors

Peter Schwabe



Joint work with Bo-Yin Yang, Shang-Yi Yang

July 11, 2012

Africacrypt 2012, Ifrane, Morocco

The SHA-3 competition

• Current situation of cryptographic hash functions:

- MD5 is broken
- SHA-1 does not even offer 80 bits of security
- SHA-2 is based on a similar design as SHA-1

The SHA-3 competition

Current situation of cryptographic hash functions:

- MD5 is broken
- SHA-1 does not even offer 80 bits of security
- SHA-2 is based on a similar design as SHA-1
- ► In 2007 NIST issued a public call for a new hash function: SHA-3
- By Oct. 2008: 64 submissions, 51 were selected as round-1 candidates
- July 2009: 14 candidates selected for round 2
- Dec. 2010: 5 finalists selected
- ▶ Final decision expected in 2012

- Blake, designed by Aumasson, Henzen, Meier, and Phan
- Grøstl, designed by Gauravaram, Knudsen, Matusiewicz, Mendel, Rechberger, Schläffer, and Thomsen
- ▶ JH, designed by Wu
- ► Keccak, designed by Bertoni, Daemen, Peeters, and Van Assche
- Skein, designed by Ferguson, Lucks, Schneier, Whiting, Bellare, Kohno, Callas, and Walker

- Blake, designed by Aumasson, Henzen, Meier, and Phan
- Grøstl, designed by Gauravaram, Knudsen, Matusiewicz, Mendel, Rechberger, Schläffer, and Thomsen
- ► JH, designed by Wu
- Keccak, designed by Bertoni, Daemen, Peeters, and Van Assche
- Skein, designed by Ferguson, Lucks, Schneier, Whiting, Bellare, Kohno, Callas, and Walker
- Requirement for all of these functions: Support output lengths of 224, 256, 384 and 512 bits
- In the following: Focus on 256-bit-output versions

- Blake, designed by Aumasson, Henzen, Meier, and Phan
- Grøstl, designed by Gauravaram, Knudsen, Matusiewicz, Mendel, Rechberger, Schläffer, and Thomsen
- ▶ JH, designed by Wu
- Keccak, designed by Bertoni, Daemen, Peeters, and Van Assche
- Skein, designed by Ferguson, Lucks, Schneier, Whiting, Bellare, Kohno, Callas, and Walker
- Requirement for all of these functions: Support output lengths of 224, 256, 384 and 512 bits
- In the following: Focus on 256-bit-output versions
- No attacks against NIST's core requirements found in any of the candidates

- Blake, designed by Aumasson, Henzen, Meier, and Phan
- Grøstl, designed by Gauravaram, Knudsen, Matusiewicz, Mendel, Rechberger, Schläffer, and Thomsen
- ▶ JH, designed by Wu
- Keccak, designed by Bertoni, Daemen, Peeters, and Van Assche
- Skein, designed by Ferguson, Lucks, Schneier, Whiting, Bellare, Kohno, Callas, and Walker
- Requirement for all of these functions: Support output lengths of 224, 256, 384 and 512 bits
- In the following: Focus on 256-bit-output versions
- No attacks against NIST's core requirements found in any of the candidates
- Important selection criterion: Performance in software

From the Announcing Request for Candidate Algorithm Nominations for a New Cryptographic Hash Algorithm (SHA–3) Family, NIST, 2007:

"At a minimum, the submitter shall state efficiency estimates for the "NIST SHA–3 Reference Platform" (specified in section 6.B) and for 8-bit processors."

From the Announcing Request for Candidate Algorithm Nominations for a New Cryptographic Hash Algorithm (SHA–3) Family, NIST, 2007:

"Two optimized implementations of the candidate algorithm shall be submitted—one implementation that is optimized for a 32-bit platform, and another for a 64-bit platform. The optimized implementations shall be specified in the ANSI C programming language."

From section 6.B: NIST Reference Platform: Wintel personal computer, with an Intel Core 2 Duo Processor, 2.4GHz clock speed, 2GB RAM, running Windows Vista Ultimate 32-bit (x86) and 64-bit (x64) Edition.

- From section 6.B: NIST Reference Platform: Wintel personal computer, with an Intel Core 2 Duo Processor, 2.4GHz clock speed, 2GB RAM, running Windows Vista Ultimate 32-bit (x86) and 64-bit (x64) Edition.
- Obvious choice for submitters and implementors: Focus on x86 and x64

- From section 6.B: NIST Reference Platform: Wintel personal computer, with an Intel Core 2 Duo Processor, 2.4GHz clock speed, 2GB RAM, running Windows Vista Ultimate 32-bit (x86) and 64-bit (x64) Edition.
- Obvious choice for submitters and implementors: Focus on x86 and x64
- How about performance on other platforms?
- How well do C compilers optimize the code?

- From section 6.B: NIST Reference Platform: Wintel personal computer, with an Intel Core 2 Duo Processor, 2.4GHz clock speed, 2GB RAM, running Windows Vista Ultimate 32-bit (x86) and 64-bit (x64) Edition.
- Obvious choice for submitters and implementors: Focus on x86 and x64
- How about performance on other platforms?
- How well do C compilers optimize the code?
- Partial answer: eBASH ECRYPT benchmarking of hash functions by Bernstein and Lange
- Benchmark all submitted implementations of all candidates on > 90 computers
- Systematically try many different compilers and compiler options for each implementation

- From section 6.B: NIST Reference Platform: Wintel personal computer, with an Intel Core 2 Duo Processor, 2.4GHz clock speed, 2GB RAM, running Windows Vista Ultimate 32-bit (x86) and 64-bit (x64) Edition.
- Obvious choice for submitters and implementors: Focus on x86 and x64
- How about performance on other platforms?
- How well do C compilers optimize the code?
- Partial answer: eBASH ECRYPT benchmarking of hash functions by Bernstein and Lange
- Benchmark all submitted implementations of all candidates on > 90 computers
- Systematically try many different compilers and compiler options for each implementation
- ▶ Portable implementations in C typically fall short in performance
- Still required: Optimized (assembly) implementations for all these platforms
- Mid-2011: Only one of the finalists optimized for ARM processors

ARM processors

- 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

ARM processors

- 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
- Our paper: How fast are the 256-bit output versions of the 5 remaining SHA-3 candidates on ARM11
- Implementations in hand-optimized assembly
- For comparison we also implemented SHA-256

ARM processors

- 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
- Our paper: How fast are the 256-bit output versions of the 5 remaining SHA-3 candidates on ARM11
- Implementations in hand-optimized assembly
- ▶ For comparison we also implemented SHA-256
- Further interpretations of the results:
 - Performance of SHA-3 candidates on a "typical" 32-bit RISC microarchitecture
 - How good are compilers at optmizing existing C implementations for a simple 32-bit architecture

- ▶ We want to run SUPERCOP (the eBASH benchmarking suite)
- Need a Linux or Unix system on an ARM11
- Need access to the CPU's cycle counter

- ▶ We want to run SUPERCOP (the eBASH benchmarking suite)
- Need a Linux or Unix system on an ARM11
- Need access to the CPU's cycle counter
- One possibility: Use ARM11 development boards (e.g., FriendlyARM, Raspberry Pi)

- ▶ We want to run SUPERCOP (the eBASH benchmarking suite)
- Need a Linux or Unix system on an ARM11
- Need access to the CPU's cycle counter
- One possibility: Use ARM11 development boards (e.g., FriendlyARM, Raspberry Pi)
- Other possibility: Use an (Android) phone
- Root the phone, install Debian in a chroot environment
- Obtain Android Linux kernel source code

- We want to run SUPERCOP (the eBASH benchmarking suite)
- Need a Linux or Unix system on an ARM11
- Need access to the CPU's cycle counter
- One possibility: Use ARM11 development boards (e.g., FriendlyARM, Raspberry Pi)
- Other possibility: Use an (Android) phone
- Root the phone, install Debian in a chroot environment
- Obtain Android Linux kernel source code
- ► Enable cycle counter through kernel module by Bernstein

- We want to run SUPERCOP (the eBASH benchmarking suite)
- Need a Linux or Unix system on an ARM11
- Need access to the CPU's cycle counter
- One possibility: Use ARM11 development boards (e.g., FriendlyARM, Raspberry Pi)
- Other possibility: Use an (Android) phone
- Root the phone, install Debian in a chroot environment
- Obtain Android Linux kernel source code
- ► Enable cycle counter through kernel module by Bernstein
- Our development environment:
 - Samsung GT i7500 Galaxy smart phone
 - GAOSP Android firmware
 - 2.6.29 Linux kernel
 - Debian running in chroot

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
 - \blacktriangleright 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
 - \blacktriangleright This input is needed one cycle earlier in the pipeline \Rightarrow "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)

- \blacktriangleright Main work: 14 rounds, each consisting of 8 evaluations of G
- \blacktriangleright Each G: 6 additions, 6 xors, 4 rotations by fixed distances
- This processes 64 bytes of input

- \blacktriangleright Main work: 14 rounds, each consisting of 8 evaluations of G
- Each G: 6 additions, 6 xors, 4 rotations by fixed distances
- This processes 64 bytes of input
- Rotations are not applied to one input but to the output
- Merge rotations of outputs with arithmetic:
 - Do not rotate output after instruction, rotate for free when the value is used as input

- Main work: 14 rounds, each consisting of 8 evaluations of G
- Each G: 6 additions, 6 xors, 4 rotations by fixed distances
- This processes 64 bytes of input
- Rotations are not applied to one input but to the output
- 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:

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

Compute:

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

and set the implicit rotation distance of a to n_1

- Main work: 14 rounds, each consisting of 8 evaluations of G
- Each G: 6 additions, 6 xors, 4 rotations by fixed distances
- This processes 64 bytes of input
- Rotations are not applied to one input but to the output
- 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:

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

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

- Main work: 14 rounds, each consisting of 8 evaluations of G
- Each G: 6 additions, 6 xors, 4 rotations by fixed distances
- This processes 64 bytes of input
- Rotations are not applied to one input but to the output
- 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:

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

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 implementation

- Main work: 10 rounds, each consisting of permutations P and Q, similar to AES
- Multiple possible implementation techniques: lookup tables, bitslicing, vector permute
- Most promising for ARM11: lookup tables (similar to AES)
- Each round, each permutation: 64 64-bit table lookups and 56 xors of 64-bit values
- This processes 64 bytes of input

Grøstl implementation

- Main work: 10 rounds, each consisting of permutations P and Q, similar to AES
- Multiple possible implementation techniques: lookup tables, bitslicing, vector permute
- Most promising for ARM11: lookup tables (similar to AES)
- Each round, each permutation: 64 64-bit table lookups and 56 xors of 64-bit values
- This processes 64 bytes of input
- Assembly implementation by Wieser: 140.17 cycles/byte
- Very well optimized but only uses 32-bit loads

Grøstl implementation

- Main work: 10 rounds, each consisting of permutations P and Q, similar to AES
- Multiple possible implementation techniques: lookup tables, bitslicing, vector permute
- Most promising for ARM11: lookup tables (similar to AES)
- Each round, each permutation: 64 64-bit table lookups and 56 xors of 64-bit values
- This processes 64 bytes of input
- Assembly implementation by Wieser: 140.17 cycles/byte
- Very well optimized but only uses 32-bit loads
- ▶ 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 implementation

- 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)
- This processes 64 bytes

JH implementation

- 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)
- This processes 64 bytes
- 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 implementation

- 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)
- This processes 64 bytes
- 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 adjacent 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 (and 55 rotates)
- This processes 128 bytes
- 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 (and 55 rotates)
- This processes 128 bytes
- 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" (round-constant injection)
- This processes 64 bytes

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" (round-constant injection)
- This processes 64 bytes
- 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" (round-constant injection)
- This processes 64 bytes
- 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	262.34 (bitslice_opt32,
		(benchmark from diablo)
Keccak	71.73	86.95 (simple32bi)
Skein	42.10	94.57 (sphlib-small v3.0)
SHA-256	26.60	39.19 (sphlib v3.0)

Details for various message lengths and quartiles in the paper.

Some remarks and conclusion

Why is every SHA-3 finalist slower than SHA-256 on ARM11?

- ▶ SHA-256 (as Blake-256) is designed for 32-bit processors
- ▶ SHA-256 uses smaller state that fits into registers (fewer spills)

Some remarks and conclusion

Why is every SHA-3 finalist slower than SHA-256 on ARM11?

- ▶ SHA-256 (as Blake-256) is designed for 32-bit processors
- ► SHA-256 uses smaller state that fits into registers (fewer spills)

How about software side channels?

- Grøstl implementation is vulnerable to timing attacks
- All other implementations run in constant time
- ► Constant-time Grøstl would be much (?) slower on ARM11

Some remarks and conclusion

Why is every SHA-3 finalist slower than SHA-256 on ARM11?

- ▶ SHA-256 (as Blake-256) is designed for 32-bit processors
- ► SHA-256 uses smaller state that fits into registers (fewer spills)

How about software side channels?

- Grøstl implementation is vulnerable to timing attacks
- All other implementations run in constant time
- ► Constant-time Grøstl would be much (?) slower on ARM11

How fast can you be with C implementations?

- Compilers don't optimize generic C implementations well enough
- Some tricks would have been possible also in C
- Then again: Writing micro-architecture-optimized code in C cannot really be the point

Results online

- All software is in the public domain and included in SUPERCOP http://bench.cr.yp.to/supercop.html
- Paper is online at http://cryptojedi.org/papers/#sha3arm
- Slides will be online http://cryptojedi.org/users/peter/#talks