
Quo vadis BLAKE?

Outline of this talk:

Specification

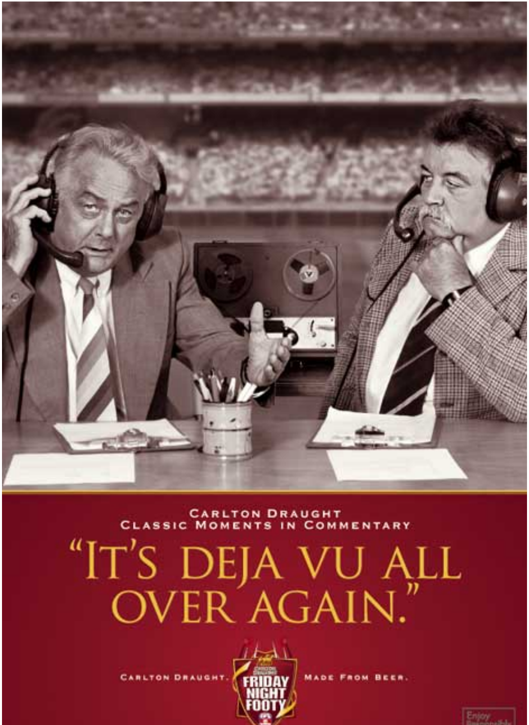
Design rationale

Security

Software performance

Hardware performance

Conclusion


CARLTON DRAUGHT
CLASSIC MOMENTS IN COMMENTARY
"IT'S DEJA VU ALL OVER AGAIN."
CARLTON DRAUGHT.
FRIDAY NIGHT FOOTY
MADE FROM BEER.

Find all info on

http://131002.net/blake/

http://en.wikipedia.org/wiki/BLAKE_(hash_function)

http://www.nist.gov/hash-competition

http://bench.cr.yp.to/results-sha3.html

http://ehash.iaik.tugraz.at/wiki/The_SHA-3_Zoo

http://xbx.das-labor.org

http://cryptography.gmu.edu/athenadb/table_view

First and Second SHA3 Conference presentations

Next talks of today and tomorrow

Etc.

The single most asked question…

The single most asked question…

Why “BLAKE”?

The single most asked question…

Why “BLAKE”?

Why BLAKE?

The single most asked question…

Why “BLAKE”?

Why BLAKE?

Why for SHA3?

PART 1: From LAKE to BLAKE

It all started with **LAKE**…

It all started with **LAKE**...

"LAKE" is the name I proposed in 2006 for what became

TCHo: A Hardware-Oriented Trapdoor Cipher

Jean-Philippe Aumasson[1,⋆], Matthieu Finiasz[2], Willi Meier[1,⋆⋆], and Serge Vaudenay[3]

[1] FHNW, Windisch, Switzerland
[2] ENSTA, Paris, France
[3] EPFL, Lausanne, Switzerland
http://lasecwww.epfl.ch/

'was refused... (but the paper was accepted to ACISP'07)

It all started with **LAKE**. . .

“LAKE” is the name I proposed in 2006 for what became

TCHo: A Hardware-Oriented Trapdoor Cipher

Jean-Philippe Aumasson[1,⋆], Matthieu Finiasz[2], Willi Meier[1,⋆⋆], and Serge Vaudenay[3]

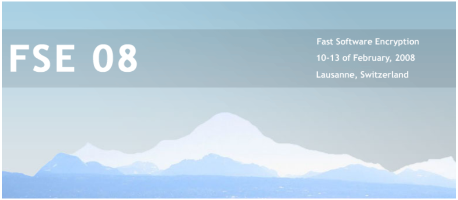
[1] FHNW, Windisch, Switzerland
[2] ENSTA, Paris, France
[3] EPFL, Lausanne, Switzerland
http://lasecwww.epfl.ch/

’was refused. . . (but the paper was accepted to ACISP’07)

Since worked hard to find this name, determined to put it on a future design. . .

2007 Oct: we submit the hash function LAKE to FSE 2008

2007 Nov: NIST announces the SHA3 competition



DEPARTMENT OF COMMERCE

National Institute of Standards and Technology

[Docket No.: 070911510–7512–01]

Announcing Request for Candidate Algorithm Nominations for a New Cryptographic Hash Algorithm (SHA–3) Family

AGENCY: National Institute of Standards and Technology, Commerce.

ACTION: Notice and request for nominations for candidate hash algorithms.

LAKE innovations: HAIFA, built-in salt, local wide-pipe

The Hash Function Family LAKE

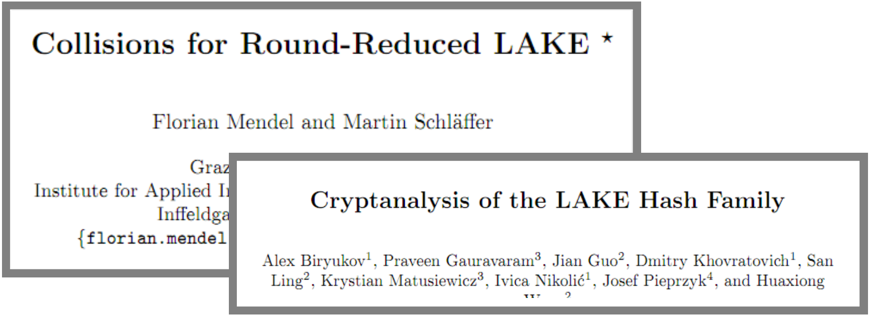
Jean-Philippe Aumasson[1*], Willi Meier[1], and Raphael C.-W. Phan[2**]

[1] FHNW, 5210 Windisch, Switzerland

[2] Electronic & Electrical Engineering, Loughborough University, LE11 3TU, United Kingdom

LAKE wasn’t good enough

Flaws in the compression, though hash unattacked


Collisions for Round-Reduced LAKE *

Florian Mendel and Martin Schläffer

Graz
Institute for Applied I
Inffeldga
{florian.mendel

Cryptanalysis of the LAKE Hash Family

Alex Biryukov[1], Praveen Gauravaram[3], Jian Guo[2], Dmitry Khovratovich[1], San Ling[2], Krystian Matusiewicz[3], Ivica Nikolić[1], Josef Pieprzyk[4], and Huaxiong

Starting the development of BLAKE after FSE (Feb ’08)

From LAKE to BLAKE…

Keep HAIFA: counter & salt, avoids length extension

Keep the local wide-pipe and global narrow-pipe

- Straightforward no-collision proof for fixed block
- Larger state allows to add redundancy, counter, salt
- Narrow-pipe attacks not a concern in practice

Keep the compression algorithm, NOT

- **Complete redesign needed**

General design philosophy:

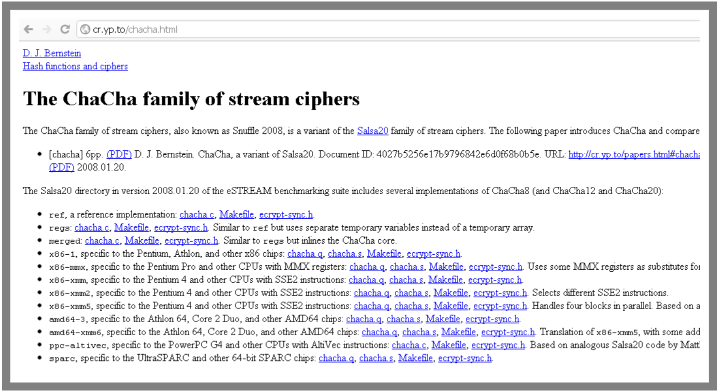
- KISS
- Think to users and implementers
- Don't optimize
- Don't reinvent the wheel

Understand the **needs** of SHA3

- Who will be the SHA3 users?
- Properties that are mandatory/desirable/superfluous?

Remember that SHA3 is an engineering competition, and not the place for experimental, untested, and inefficient designs (however interesting and technically deep)

BLAKE's core: a robust, previously-analyzed design



cr.yp.to/chacha.html

D. J. Bernstein
Hash functions and ciphers

The ChaCha family of stream ciphers

The ChaCha family of stream ciphers, also known as Snuffle 2008, is a variant of the Salsa20 family of stream ciphers. The following paper introduces ChaCha and compare

- [chacha] 6pp. (PDF) D. J. Bernstein. ChaCha, a variant of Salsa20. Document ID: 4027b5256e17b9796842e6d0f68b0b5e. URL: http://cr.yp.to/papers.html#chacha (PDF) 2008.01.20.

The Salsa20 directory in version 2008.01.20 of the eSTREAM benchmarking suite includes several implementations of ChaCha8 (and ChaCha12 and ChaCha20):

- `ref`, a reference implementation: chacha.c, Makefile, ecrypt-sync.h.
- `regs`: chacha.c, Makefile, ecrypt-sync.h. Similar to `ref` but uses separate temporary variables instead of a temporary array.
- `merged`: chacha.c, Makefile, ecrypt-sync.h. Similar to `regs` but inlines the ChaCha core.
- `x86-1`, specific to the Pentium, Athlon, and other x86 chips: chacha.q, chacha.s, Makefile, ecrypt-sync.h.
- `x86-mmx`, specific to the Pentium Pro and other CPUs with MMX registers: chacha.q, chacha.s, Makefile, ecrypt-sync.h. Uses some MMX registers as substitutes for
- `x86-xmm`, specific to the Pentium 4 and other CPUs with SSE2 instructions: chacha.q, chacha.s, Makefile, ecrypt-sync.h.
- `x86-xmm2`, specific to the Pentium 4 and other CPUs with SSE2 instructions: chacha.q, chacha.s, Makefile, ecrypt-sync.h. Selects different SSE2 instructions.
- `x86-xmm5`, specific to the Pentium 4 and other CPUs with SSE2 instructions: chacha.q, chacha.s, Makefile, ecrypt-sync.h. Handles four blocks in parallel. Based on a
- `amd64-3`, specific to the Athlon 64, Core 2 Duo, and other AMD64 chips: chacha.q, chacha.s, Makefile, ecrypt-sync.h.
- `amd64-xmm6`, specific to the Athlon 64, Core 2 Duo, and other AMD64 chips: chacha.q, chacha.s, Makefile, ecrypt-sync.h. Translation of `x86-xmm5`, with some add
- `ppc-altivec`, specific to the PowerPC G4 and other CPUs with AltiVec instructions: chacha.c, Makefile, ecrypt-sync.h. Based on analogous Salsa20 code by Matt
- `sparc`, specific to the UltraSPARC and other 64-bit SPARC chips: chacha.q, chacha.s, Makefile, ecrypt-sync.h.

ChaCha's core is a strong well-analyzed 4-word map

After several prototype designs, decided to extend the ChaCha permutation to form BLAKE's core

Previous project (FSE'08)

New Features of Latin Dances:
Analysis of Salsa, ChaCha, and Rumba

Jean-Philippe Aumasson[1], Simon Fischer[1], Shahram Khazaei[2], Willi Meier[1], and Christian Rechberger[3]

[1] FHNW, Windisch, Switzerland
[2] EPFL, Lausanne, Switzerland
[3] IAIK, Graz, Austria

Amazed at ChaCha/Salsa20's simplicity and efficiency

Intrinsic $4\times$ parallelism, faster diffusion in ChaCha

Motivations for ARX:

- Performance tradeoff HW/SW
- Easy to implement
- Fast confusion/diffusion

ChaCha's simplistic "quarterround" function

Bijective transform of four 32-bit words (a,b,c,d)

$$
\begin{array}{ll}
a \mathrel{+}= b & d = (a \oplus d) \lll 16 \\
c \mathrel{+}= d & b = (b \oplus c) \lll 12 \\
\\
a \mathrel{+}= b & d = (a \oplus d) \lll 8 \\
c \mathrel{+}= d & b = (b \oplus c) \lll 7
\end{array}
$$

BLAKE-256's **G** function

Repeated 112 times in BLAKE-256 (32-bit words)

$$
\begin{array}{ll}
a \mathrel{+}= m_i \oplus k_i & \\
a \mathrel{+}= b & d = (a \oplus d) \ggg 16 \\
c \mathrel{+}= d & b = (b \oplus c) \ggg 12 \\
a \mathrel{+}= m_j \oplus k_j & \\
a \mathrel{+}= b & d = (a \oplus d) \ggg 8 \\
c \mathrel{+}= d & b = (b \oplus c) \ggg 7
\end{array}
$$

BLAKE-512's **G** function

Repeated 128 times in BLAKE-512 (64-bit words)

$$
\begin{array}{ll}
a \mathrel{+}= m_i \oplus k_i & \\
a \mathrel{+}= b & d = (a \oplus d) \ggg 32 \\
c \mathrel{+}= d & b = (b \oplus c) \ggg 25 \\
a \mathrel{+}= m_j \oplus k_j & \\
a \mathrel{+}= b & d = (a \oplus d) \ggg 16 \\
c \mathrel{+}= d & b = (b \oplus c) \ggg 11
\end{array}
$$

Counting ARX ops:

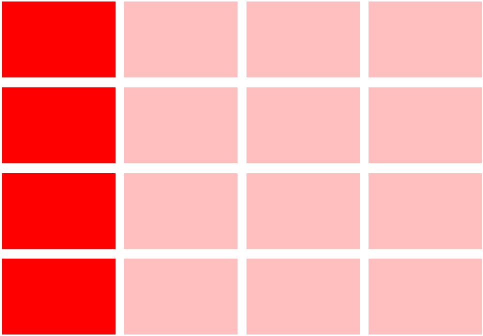
	BLAKE-256	BLAKE-512
Word	32-bit	64-bit
$+$	672	768
$\oplus$	672	768
$\lll$	448	512
Total	1792	2048
Ops/word	112	128
Ops/byte	3.5	2.0

BLAKE's 4×4 internal state

Initialized with chaining value, salt, counter, constants

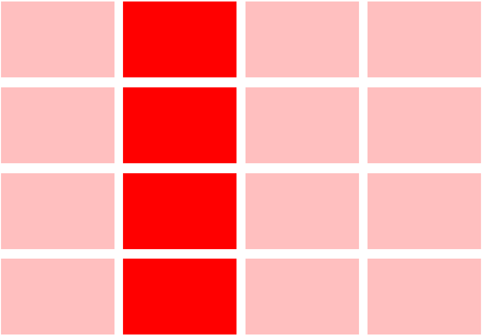
A BLAKE round:

Apply the **G** function to each column



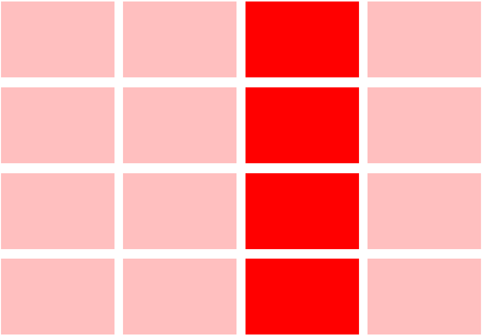
A BLAKE round:

Apply the **G** function to each column



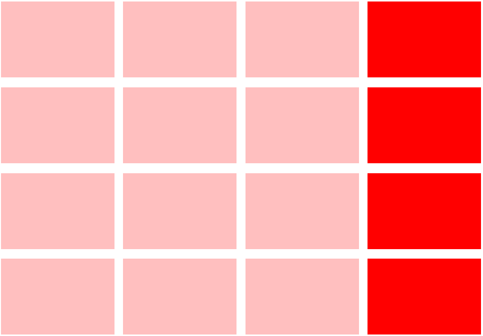
A BLAKE round:

Apply the **G** function to each column



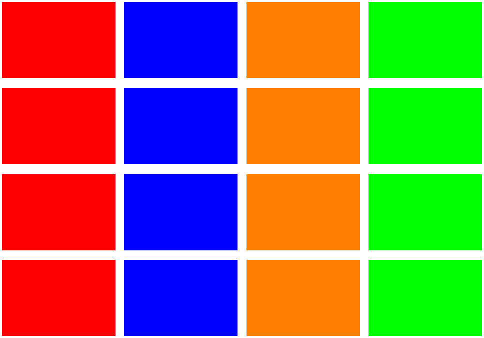
A BLAKE round:

Apply the **G** function to each column



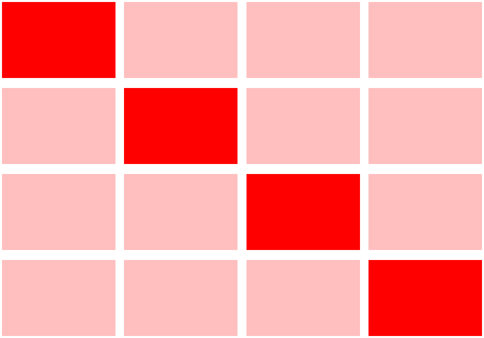
A BLAKE round:

Apply the **G** function to each column (in parallel)



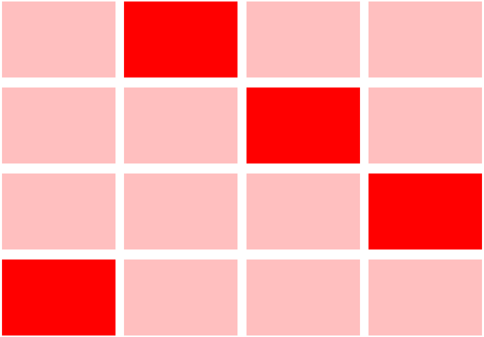
A BLAKE round:

Apply the **G** function to each diagonal



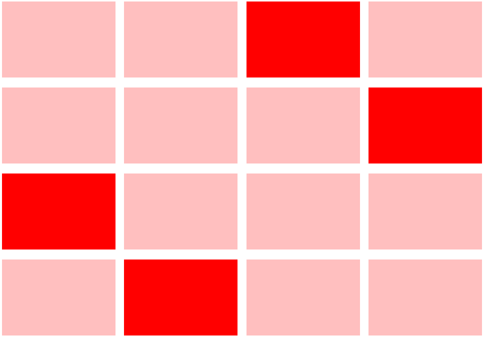
A BLAKE round:

Apply the **G** function to each diagonal



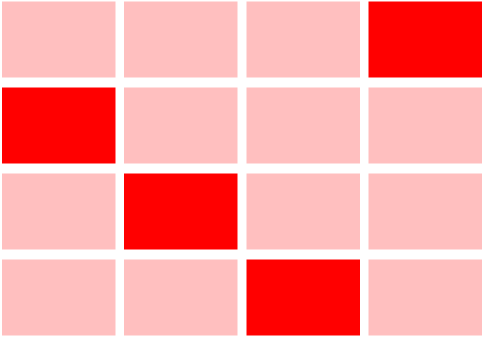
A BLAKE round:

Apply the **G** function to each diagonal



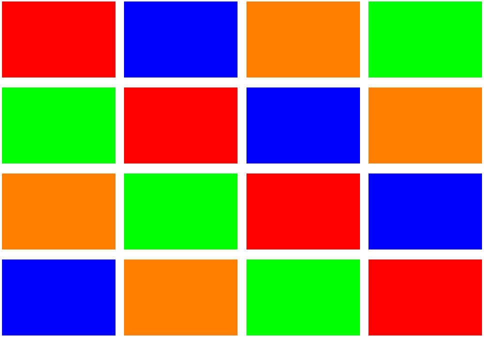
A BLAKE round:

Apply the **G** function to each diagonal



A BLAKE round:

Apply the **G** function to each diagonal (in parallel)

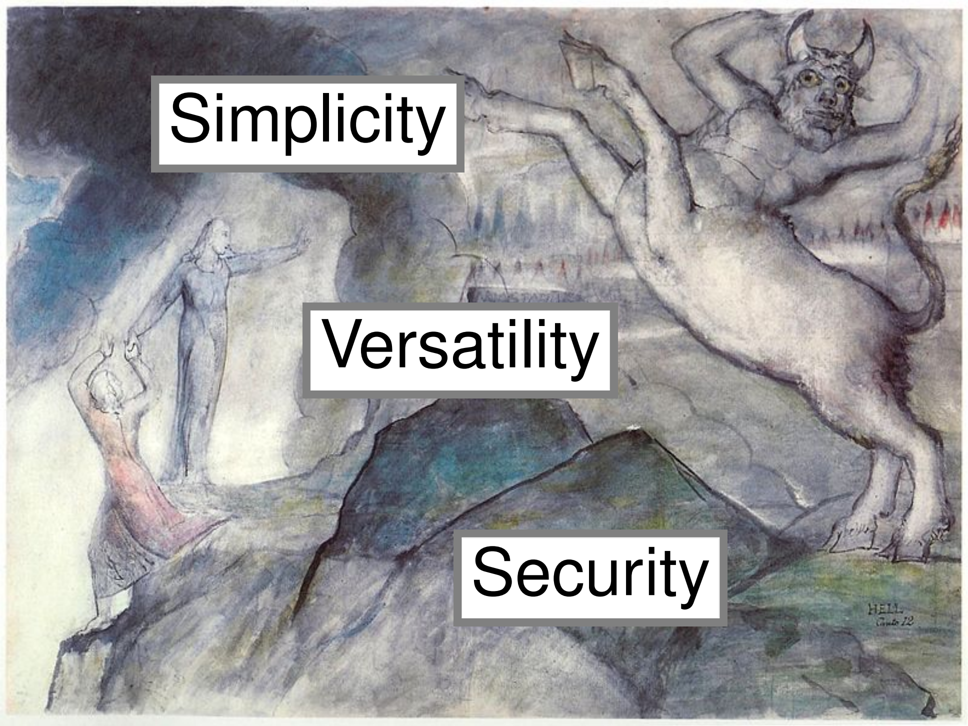


Why the name “BLAKE”?

- Expresses the LAKE legacy
- Can be understood as “Better LAKE” (unintentional)
- Short, simple to write and to correctly pronounce
- No negative meaning or translation
- Reference to William Blake

More popular Blake's (according to Google):

PART 2: BLAKE's unique qualities


Simplicity
Versatility
Security

Simplicity

Easy-to-understand specs

- Simplified HAIFA mode
- Familiar 4×4 state representation
- A single core function: **G**
- Only ops used are standard $+$, $\oplus$, $\lll$
- Repetition of just 3 lines of code

$\boxed{\text{Simplicity}}$

Easy-to-implement

- Clean version in 185 lines of C
- Small “attack surface” for coding errors
- Only need implement **G**, plus administrative code
- Reduces production costs (debug time, etc.)

“simple and clear design”, in NIST 2nd Round Report

```
void blake256_compress( state *S, const u8 *block ) {

  u32 v[16], m[16], i;
#define ROT(x,n) (((x)<<(32-n))|( (x)>>(n)))
#define G(a,b,c,d,e) \
  v[a] += (m[sigma[i][e]] ^ cst[sigma[i][e+1]]) + v[b]; \
  v[d] = ROT( v[d] ^ v[a],16); \
  v[c] += v[d]; \
  v[b] = ROT( v[b] ^ v[c],12); \
  v[a] += (m[sigma[i][e+1]] ^ cst[sigma[i][e]])+v[b]; \
  v[d] = ROT( v[d] ^ v[a], 8); \
  v[c] += v[d]; \
  v[b] = ROT( v[b] ^ v[c], 7);

  for(i=0; i<16;++i) m[i] = U8TO32(block + i*4);
  for(i=0; i< 8;++i) v[i] = S->h[i];
  v[ 8] = S->s[0] ^ 0x243F6A88; v[12] = 0xA4093822;
  v[ 9] = S->s[1] ^ 0x85A308D3; v[13] = 0x299F31D0;
  v[10] = S->s[2] ^ 0x13198A2E; v[14] = 0x082EFA98;
  v[11] = S->s[3] ^ 0x03707344; v[15] = 0xEC4E6C89;
  if (S->nullt == 0) {
    v[12] ^= S->t[0]; v[13] ^= S->t[0];
    v[14] ^= S->t[1]; v[15] ^= S->t[1];
  }
```

```
#define G(a,b,c,d,e) \
  v[a] += (m[sigma[i][e]] ^ cst[sigma[i][e+1]]) + v[b]; \
  v[d] = ROT( v[d] ^ v[a],16); \
  v[c] += v[d]; \
  v[b] = ROT( v[b] ^ v[c],12); \
  v[a] += (m[sigma[i][e+1]] ^ cst[sigma[i][e]])+v[b]; \
  v[d] = ROT( v[d] ^ v[a], 8); \
  v[c] += v[d]; \
  v[b] = ROT( v[b] ^ v[c], 7);

  for(i=0; i<16;++i) m[i] = U8TO32(block + i*4);
  for(i=0; i< 8;++i) v[i] = S->h[i];
  v[ 8] = S->s[0] ^ 0x243F6A88; v[12] = 0xA4093822;
  v[ 9] = S->s[1] ^ 0x85A308D3; v[13] = 0x299F31D0;
  v[10] = S->s[2] ^ 0x13198A2E; v[14] = 0x082EFA98;
  v[11] = S->s[3] ^ 0x03707344; v[15] = 0xEC4E6C89;
  if (S->nullt == 0) {
    v[12] ^= S->t[0]; v[13] ^= S->t[0];
    v[14] ^= S->t[1]; v[15] ^= S->t[1];
  }
  for(i=0; i<14; ++i) {
    G( 0, 4, 8,12, 0);
    G( 1, 5, 9,13, 2);
    G( 2, 6,10,14, 4);
```

```
v[c] += v[d]; \
v[b] = ROT( v[b] ^ v[c],12); \
v[a] += (m[sigma[i][e+1]] ^ cst[sigma[i][e]])+v[b]; \
v[d] = ROT( v[d] ^ v[a], 8); \
v[c] += v[d]; \
v[b] = ROT( v[b] ^ v[c], 7);

for(i=0; i<16;++i) m[i] = U8TO32(block + i*4);
for(i=0; i< 8;++i) v[i] = S->h[i];
v[ 8] = S->s[0] ^ 0x243F6A88; v[12] = 0xA4093822;
v[ 9] = S->s[1] ^ 0x85A308D3; v[13] = 0x299F31D0;
v[10] = S->s[2] ^ 0x13198A2E; v[14] = 0x082EFA98;
v[11] = S->s[3] ^ 0x03707344; v[15] = 0xEC4E6C89;
if (S->nullt == 0) {
  v[12] ^= S->t[0]; v[13] ^= S->t[0];
  v[14] ^= S->t[1]; v[15] ^= S->t[1];
}
for(i=0; i<14; ++i) {
  G( 0, 4, 8,12, 0);
  G( 1, 5, 9,13, 2);
  G( 2, 6,10,14, 4);
  G( 3, 7,11,15, 6);
  G( 3, 4, 9,14,14);
  G( 2, 7, 8,13,12);
```

```
v[d] = ROT( v[d] ^ v[a], 8); \
v[c] += v[d]; \
v[b] = ROT( v[b] ^ v[c], 7);

for(i=0; i<16;++i) m[i] = U8TO32(block + i*4);
for(i=0; i< 8;++i) v[i] = S->h[i];
v[ 8] = S->s[0] ^ 0x243F6A88; v[12] = 0xA4093822;
v[ 9] = S->s[1] ^ 0x85A308D3; v[13] = 0x299F31D0;
v[10] = S->s[2] ^ 0x13198A2E; v[14] = 0x082EFA98;
v[11] = S->s[3] ^ 0x03707344; v[15] = 0xEC4E6C89;
if (S->nullt == 0) {
  v[12] ^= S->t[0]; v[13] ^= S->t[0];
  v[14] ^= S->t[1]; v[15] ^= S->t[1];
}
for(i=0; i<14; ++i) {
  G( 0, 4, 8,12, 0);
  G( 1, 5, 9,13, 2);
  G( 2, 6,10,14, 4);
  G( 3, 7,11,15, 6);
  G( 3, 4, 9,14,14);
  G( 2, 7, 8,13,12);
  G( 0, 5,10,15, 8);
  G( 1, 6,11,12,10);
}
```

```
  for(i=0; i<16;++i) m[i] = U8TO32(block + i*4);
  for(i=0; i< 8;++i) v[i] = S->h[i];
  v[ 8] = S->s[0] ^ 0x243F6A88; v[12] = 0xA4093822;
  v[ 9] = S->s[1] ^ 0x85A308D3; v[13] = 0x299F31D0;
  v[10] = S->s[2] ^ 0x13198A2E; v[14] = 0x082EFA98;
  v[11] = S->s[3] ^ 0x03707344; v[15] = 0xEC4E6C89;
  if (S->nullt == 0) {
    v[12] ^= S->t[0]; v[13] ^= S->t[0];
    v[14] ^= S->t[1]; v[15] ^= S->t[1];
  }

  for(i=0; i<14; ++i) {
    G( 0, 4, 8,12, 0);
    G( 1, 5, 9,13, 2);
    G( 2, 6,10,14, 4);
    G( 3, 7,11,15, 6);
    G( 3, 4, 9,14,14);
    G( 2, 7, 8,13,12);
    G( 0, 5,10,15, 8);
    G( 1, 6,11,12,10);
  }

  for(i=0; i<16;++i) S->h[i%8] ^= v[i];
  for(i=0; i<8 ;++i) S->h[i] ^= S->s[i%4];
}
```

Versatility

BLAKE is not optimized for any specific platform

- 32- and 64-bit versions
- 256-bit digest may be produced by truncation of BLAKE-512
- Rotations multiple of 8 to simplify 8- and 16-bit implementations
- HW-friendly structure
  - Single building block **G** allows compact impl
  - Straightforward parallelism

BLAKE can be compact in FPGA

A few days ago at the ECRYPT2 Hash 2011 Workshop:

Kerckhof et al., *Compact FPGA Implementations of the Five SHA-3 Finalists*:

On Virtex 6:

		BLAKE	Grøstl	JH	Keccak	Skein	AES
Properties	Input block message size	512	512	512	1088	256	128
	Clock cycles per block	1182	176	688	2137	230	44
	Clock cycles overhead (pre/post)	12/8	122	16/20	17/16	5/234	8/0
Area	Number of LUTs	417	907	789	519	770	658
	Number of Registers	211	566	411	429	158	364
	Number of Slices	117	285	240	144	240	205
	Frequency (MHz)	274	280	288	250	160	222
	Throughput (Mbit/s)	105	815	214	128	179	646
Timing	Number of LUTs	500	966	1034	610	1039	845
	Number of Registers	284	571	463	533	506	524
	Number of Slices	175	293	304	188	291	236
	Frequency (MHz)	347	330	299	285	200	250
	Throughput (Mbit/s)	132	960	222	145	223	727

BLAKE is hardware-friendly

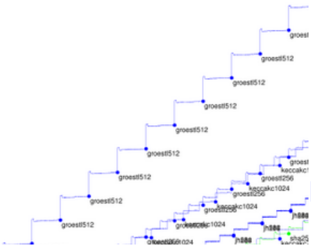
A few days ago at the ECRYPT2 Hash 2011 Workshop:

Homsirikamol, Rogawski, Gaj, *Comparing Hardware Performance of Round 3 SHA-3 Candidates using Multiple Hardware Architecture in Xilinx and Altera FPGAs*:

"BLAKE is the algorithm with the highest flexibility, and the largest number of potential architectures. It can be easily folded horizontally and vertically by factors of two and four. It can also be easily pipelined even in the folded architectures. It is also the only algorithm that has a relatively efficient architecture that is smaller than the basic iterative architecture of SHA-2. Finally, BLAKE is the only algorithm that can benefit substantially from using embedded block memories of both Xilinx and Altera FPGAs."

BLAKE is often faster than SHA2 in SW

amd64, 2833MHz, Intel Core 2 Quad Q9550 (10677), 2008, berlekamp, supercop-20110508



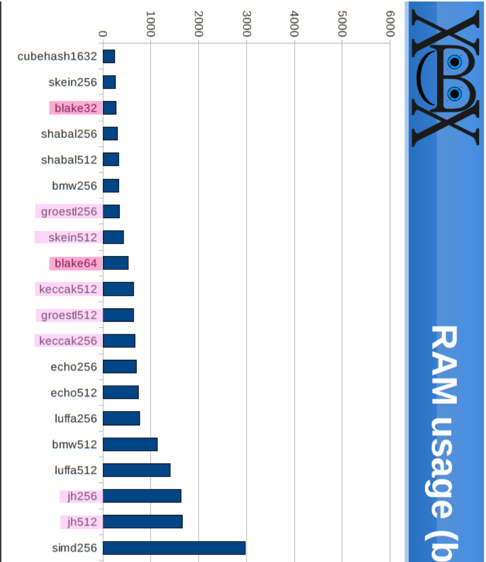
Cycles/byte for long messages			
quartile	median	quartile	hash
6.47	6.48	6.51	skein512
6.84	6.85	6.89	blake64
7.50	7.55	7.56	skein1024
7.65	7.70	7.78	blake512
7.68	7.77	7.80	blake32
7.74	7.90	8.16	skein256
8.65	8.82	9.05	blake256
10.24	10.26	10.30	sha512
10.26	10.27	10.29	sha384
10.89	10.90	10.93	keccakc448
11.65	11.69	11.72	keccakc512
12.59	12.64	12.69	keccak
15.14	15.26	15.71	sha256

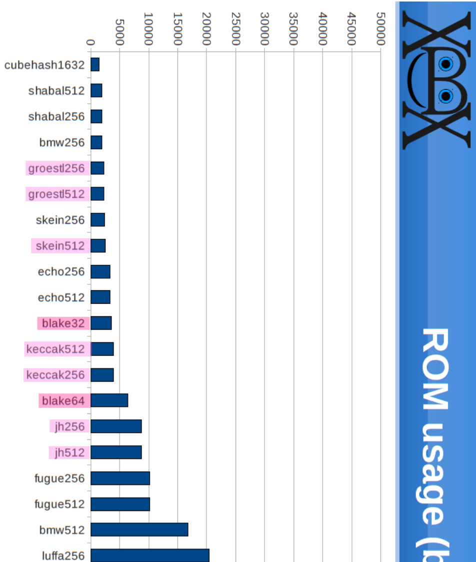
Cycles/byte for 4096 bytes			
quartile	median	quartile	hash
6.65	6.65	6.66	skein512
7.14	7.15	7.16	blake64
7.85	7.86	7.86	skein1024
7.91	7.95	7.95	blake32
8.00	8.02	8.05	blake512
7.99	8.05	8.17	skein256
8.99	9.03	9.12	blake256
10.79	10.80	10.80	sha384
10.80	10.81	10.81	sha512
11.39	11.40	11.41	keccakc448
12.15	12.16	12.18	keccakc512
13.10	13.12	13.13	keccak
15.72	15.73	15.93	sha256

Speedup from SSE (2 – 4.1) and XOP instructions, but very fast without (cf. SPHLIB code)

BLAKE is low-memory on microcontrollers

On 8-bit ATmega1281, from Wenzel-Benner 2010 slides





NIST does not disagree

In the 2nd Round Report:

“BLAKE is among the top performers in software across most platforms for long messages. BLAKE-32 is the best performer on software platforms for very short message”

“This flexibility allows cost-effective tradeoffs in area usage, with limited impact on the throughput-to-area ratio”

Security

Plenty of cryptanalysis:

Dunkelman, Khovratovich (Hash 2011)
A. Leurent, Meier, Mendel, Mouha, Phan, Sasaki, Susil (Hash 2011)
Biryukov, Nikolic, Roy (FSE 2011)
Ming, Qiang, Zeng (ICCIS 2010)
Turan, Uyan (2nd SHA3 Conf)
Vidali, Nose, Pasalic (IPL 110(14-15))
Su, Wu, Dong (ePrint 2010/355)
A. Guo, Knellwolf, Mtusiewicz, Meier (FSE 2010)
Guo, Matusiewicz (WEWoRC 2009)
Ji, Liangyu (ePrint 2009/238)

Security

Best attack on the compression function:

- Boomerang distinguisher by Biryukov/Nikolic/Roy
- 7 rounds in 2^{232} (BLAKE-256)

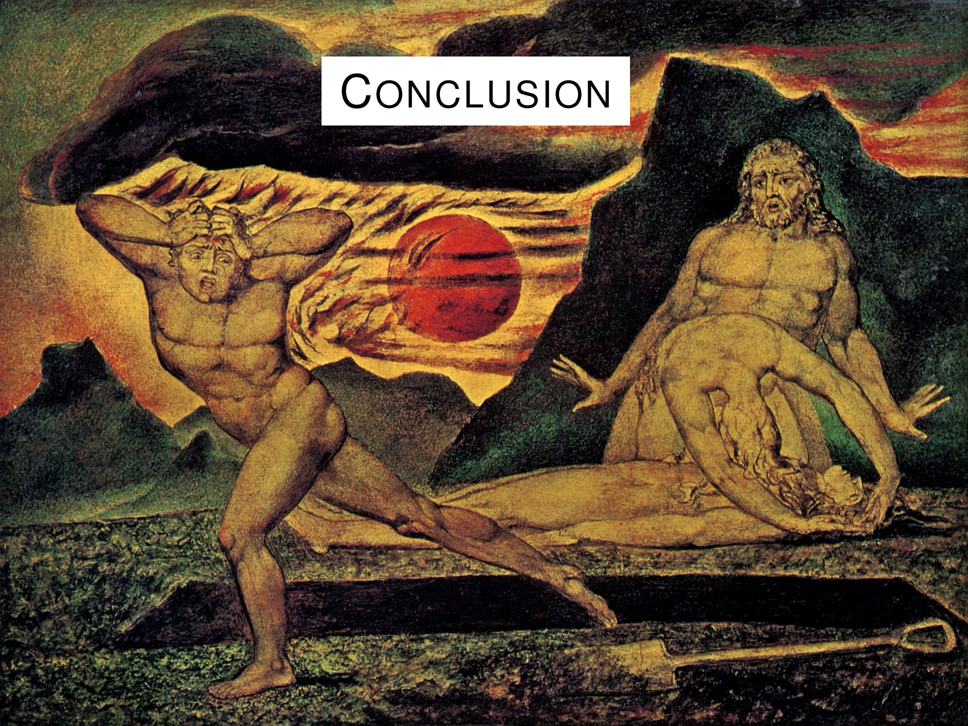
Best attack on the hash function:

- Preimage attack by Ji/Liangyu
- 2.5 rounds in 2^{241}, 2^{481}

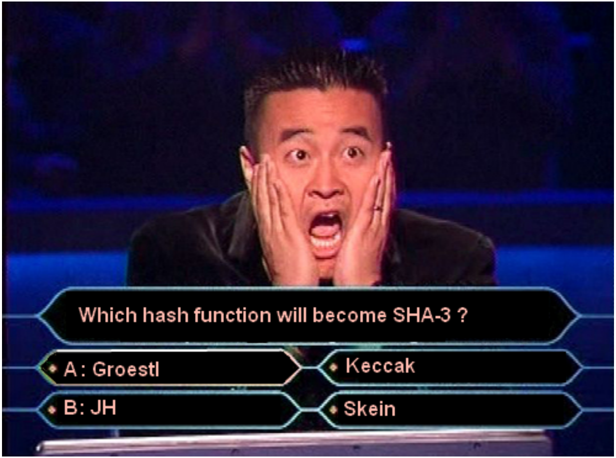
14 rounds in BLAKE-256

Security margin compares favorably with other finalists

CONCLUSION



Why BLAKE for SHA3?

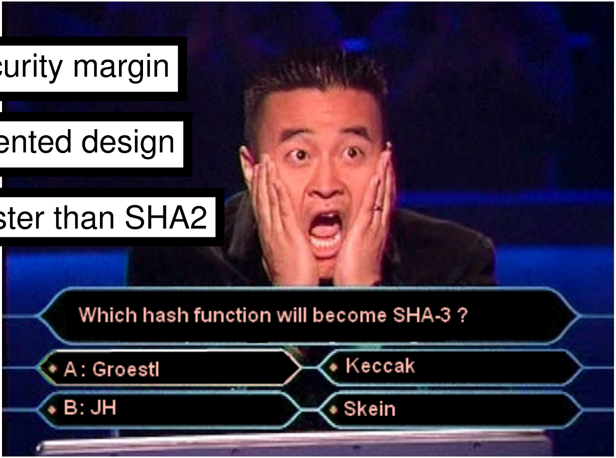


Why BLAKE for SHA3?

High security margin

User-oriented design

Often faster than SHA2



Why BLAKE for SHA3?

High security margin

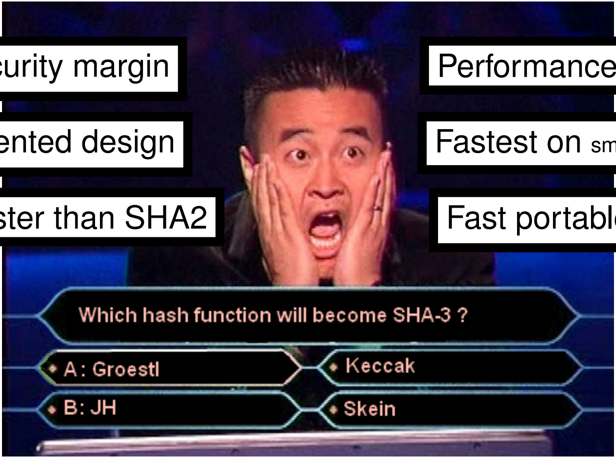
User-oriented design

Often faster than SHA2

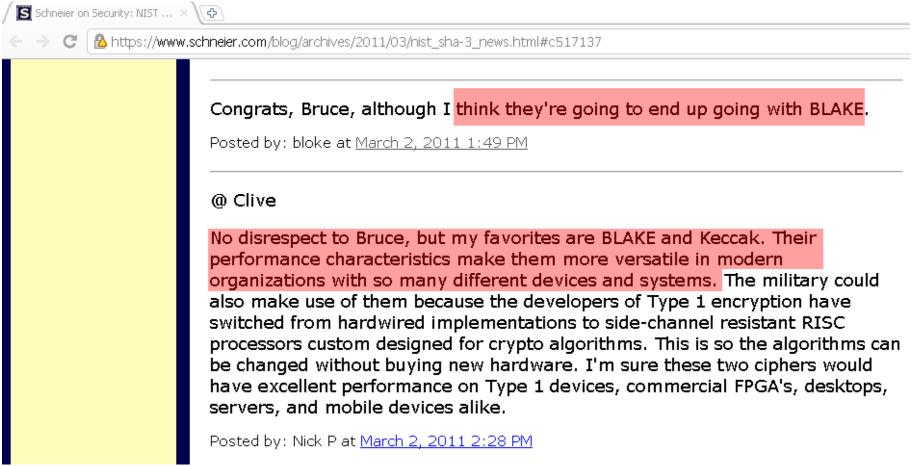
Performance versatile

Fastest on small messages

Fast portable C code



Even Schneier's blog commenters like BLAKE!


Schneier on Security: NIST ...

https://www.schneier.com/blog/archives/2011/03/nist_sha-3_news.html#c517137

Congrats, Bruce, although I think they're going to end up going with BLAKE.

Posted by: bloke at March 2, 2011 1:49 PM

@ Clive

No disrespect to Bruce, but my favorites are BLAKE and Keccak. Their performance characteristics make them more versatile in modern organizations with so many different devices and systems. The military could also make use of them because the developers of Type 1 encryption have switched from hardwired implementations to side-channel resistant RISC processors custom designed for crypto algorithms. This is so the algorithms can be changed without buying new hardware. I'm sure these two ciphers would have excellent performance on Type 1 devices, commercial FPGA's, desktops, servers, and mobile devices alike.

Posted by: Nick P at March 2, 2011 2:28 PM

Advertisement

Recent implementations of BLAKE

Python (by Larry Bugbee)

`http://tinyurl.com/pyblake`

```python
def G(a, b, c, d, i):
    va = v[a]   # it's faster to deref and reref later
    vb = v[b]
    vc = v[c]
    vd = v[d]

    sri  = SIGMA[round][i]
    sri1 = SIGMA[round][i+1]

    va = ((va + vb) + (m[sri] ^ cxx[sri1]) ) & MASK
    x  =  vd ^ va
    vd = (x >> rot1) | ((x << (WORDBITS-rot1)) & MASK)
    vc = (vc + vd) & MASK
    x  =  vb ^ vc
    vb = (x >> rot2) | ((x << (WORDBITS-rot2)) & MASK)

    va = ((va + vb) + (m[sri1] ^ cxx[sri]) ) & MASK
    x  =  vd ^ va
    vd = (x >> rot3) | ((x << (WORDBITS-rot3)) & MASK)
    vc = (vc + vd) & MASK
    x  =  vb ^ vc
    vb = (x >> rot4) | ((x << (WORDBITS-rot4)) & MASK)

    v[a] = va
    v[b] = vb
    v[c] = vc
    v[d] = vd
```

PHP (by Danniel Correa)

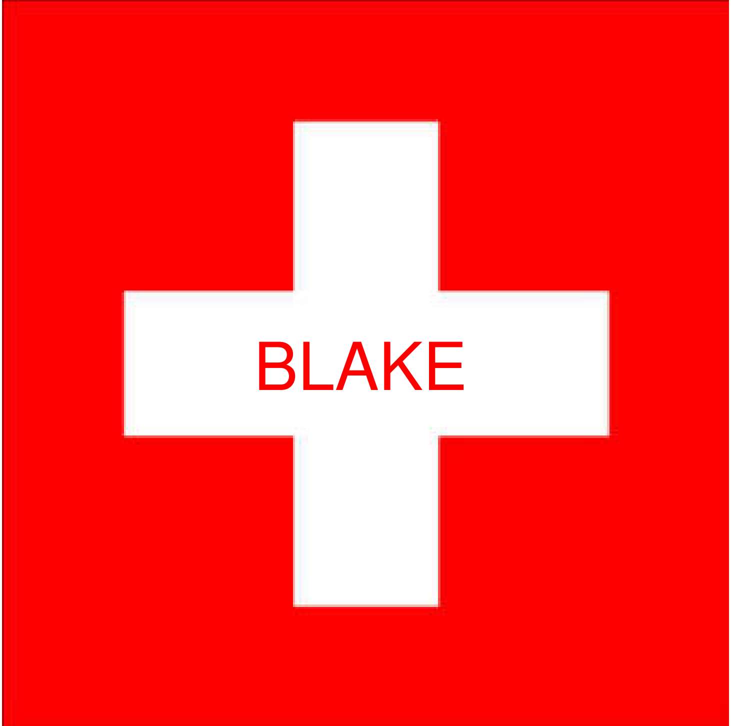
`http://tinyurl.com/phpblake1`

`http://tinyurl.com/phpblake2`

```php
<?php
echo blake('sinfocol', BLAKE_224) . PHP_EOL;
echo blake('sinfocol', BLAKE_256) . PHP_EOL;
echo blake('sinfocol', BLAKE_384) . PHP_EOL;
echo blake('sinfocol', BLAKE_512) . PHP_EOL;
$blake = blake_init(BLAKE_256, '0123456789abcdef');
blake_update($blake, 'sinfocol');
echo blake_final($blake);
/* Expected output
1427375d4a16cc70ab4155a7c721f975f92867aa53703ccd1c8f5a4b
be2f4e21cf62f1e98ba6800a73a8b887e8c69e9fbe914d64c769299b111c8974
f41a32404f454bf925b16f7b38bfd8e1910cbd31100a4e7a4ec6cbb54115ea2c028
3ec1d1ba3dfbc3ac553f5d8ad5e6c34de7b449cc9ed04b0453f9fa859f80f47e994
a7cae55a3d0f5235ae2d0e2c74ce469d855d11561a5326e46e6d7c9c4d319681
*/
```

SCREENSHOT

```
C:\>php -r "echo blake('sinfocol', BLAKE_224);"
1427375d4a16cc70ab4155a7c721f975f92867aa53703ccd1c8f5a4b
C:\>php -r "echo blake('sinfocol', BLAKE_256);"
be2f4e21cf62f1e98ba6800a73a8b887e8c69e9fbe914d64c769299b111c
C:\>php -r "echo blake('sinfocol', BLAKE_384);"
f41a32404f454bf925b16f7b38bfd8e1910cbd31100a4e7a4ec6cbb54115
04f6ddf14a1884cb
C:\>php -r "echo blake('sinfocol', BLAKE_512);"
3ec1d1ba3dfbc3ac553f5d8ad5e6c34de7b449cc9ed04b0453f9fa859f80
66b203b0d335867b4cece8c7a0dfa5092e17b1271a5a7e70
```


BLAKE